DELFT UNIVERSITY OF TECHNOLOGY

MASTERS THESIS

Online state migration in modern stream processing engines

Author:
Theodoros VENETI

Supervisor:
Dr. Asterios KATSIFODIMOS

Daily Supervisor:
Georgios SIACHAMIS

Daily Co-Supervisor:
Kyriakos PSARAKIS

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

in the

Web Information Systems Group
Software Technology

October 20, 2023

Declaration of Authorship

I, Theodoros VENETI, declare that this thesis titled, “Online state migration in modern stream processing engines” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: [illegible]

Date: 20/10/2023

DELFT UNIVERSITY OF TECHNOLOGY

Abstract

Electrical Engineering, Mathematics and Computer Science
Software Technology

Master of Science

Online state migration in modern stream processing engines

by Theodoros VENETI

Stream Processing Engines (SPEs) are called upon to help solve problems around big and volatile data, while satisfying the needs for near real-time processing. In order for such systems to be considered effective solutions to such problems at scale, efficient elasticity and non dataflow-disturbing reconfiguration operations within are a necessity. To that end, we visit the problem of online state migration, as the biggest obstacle in achieving such a desired behaviour, in SPEs that support stateful functions. We make an attempt to formally define the problem and associated sub-tasks, compare existing solutions and identify key aspects, as well as design and implement our own solution. Our testing shows that the lazy-fetch online state migration process proposed, outperforms a simple baseline state migration design by orders of magnitude in end-to-end latency observed, scales much better under increased workloads and relies on consistent design concepts to claim exactly-once semantics.

Acknowledgements

This work has been compiled thanks to the invaluable help of my thesis supervisor Dr. Asterios Katsifodimos, as well as daily supervisors Georgios Siachamis and Kyriakos Psarakis who assisted and guided me through the entire process. Their ideas, insights and extended knowledge on the topic were paramount to achieving positive results in the present work.

Moreover, I would like to thank my friends and family, for their unconditional support and love, throughout the duration of the thesis.

Contents

List of Figures

List of Tables

List of Abbreviations

SPE	**S**tream **P**rocessing **E**engine

1 Introduction

Nowadays, data processing has taken the place of what was formerly known as big data processing. Pushing the need for systems available of executing long-running jobs in a scalable and efficient manner, while the data itself are typically characterized by the 4 V's, meaning they are varied, fast, big in size and accurate as stated by Sakr, Liu, and Fayoumi, 2013. Towards satisfying all of the above, with relatively low latency, processing engines that handle data either in batches or continuous streams have been proposed and studied. In the first case, part of the data ingestion is bundled together into a batch and processing happens on the batch as a whole as if it was a single record, while on the latter the input is handled in a continuous manner with processing taking place as data arrive. Such systems have evolved way past the state of being solely academic endeavours are actually seeing production deployment offering correct executions and handling failures across multiple nodes, as the work by De Matteis and Mencagli, 2017 finds.

In an effort to simplify the representation of a big-data processing system, Zhang, Soto, and Markl, 2022 suggest modelling in the form of a tree, where information flows from the roots (input data) onto intermediate nodes (operators, meaning processing units needed for the analysis at hand) and finally to the leaves (analysis results). In this way, data "flows" through the system continuously starting from our input and into the operators which create intermediate results before being (usually) processed by more operators, until all needed processing has taken place and results are show on the output. In order for an implementation of such a design to satisfy all the previously stated requirements, it has to provide certain consistency, fault-tolerance, scalability and flexibility guarantees.

The impressive data-processing capabilities of such systems has lead to more broad applications and the necessity for more programming flexibility and implementation support. To that end, an extension to the seemingly simple case of *stateless* operators, where the output is only computed as a direct result of processing applied on the input of the unit, is being used. Called *stateful* operators, they carry with them the ability to store state, effectively giving them the ability of memory throughout their processing lifecycle, which can be used for appropriate analysis tasks. These include, but are not limited to, similarity joins, sorting or aggregation tasks such as group, min, max or average. Although the presence of state allows for a slew of additional implementation choices and features, they pose a great challenge in the previously stated requirements of the systems that support them. With state being any intermediate result necessarily stored in an operator for subsequent processing, potential problems can occur:

- Consistency: Updates on the state need to be atomic and consistent with the analysis being carried out.
- Fault-tolerance: State needs to be persistent in a way that guarantees correct version restore in case of failures.
- Flexibility: State can take any form or structure needed for the computations at hand.

- Scalability: State needs to be transferable between operators, workers or nodes in an efficient manner.

This work focuses specifically on the very last point made concerning how state can be transferred, called state migration, in a way that is fast, consistent and does not affect the dataflow throughout the system. Not disturbing the dataflow is especially important in stream processing systems, which unlike batch processing ones, provide a long-running (potentially never ending) processing job that continuously consumes data and generates results. It is paramount then, that in order to achieve high availability and low response times, the data flowing through the system are not affected by internal management processes of the system, to the extent that such is possible. This task has proven to be particularly challenging, with a blaring indication being the fact that the most widely adopted system in the industry for some time now, Apache Flink by Katsifodimos and Schelter, 2016 chooses to not deal with the problem, but rather performs a complete halt of the dataflow and restarts under the new configuration structure.

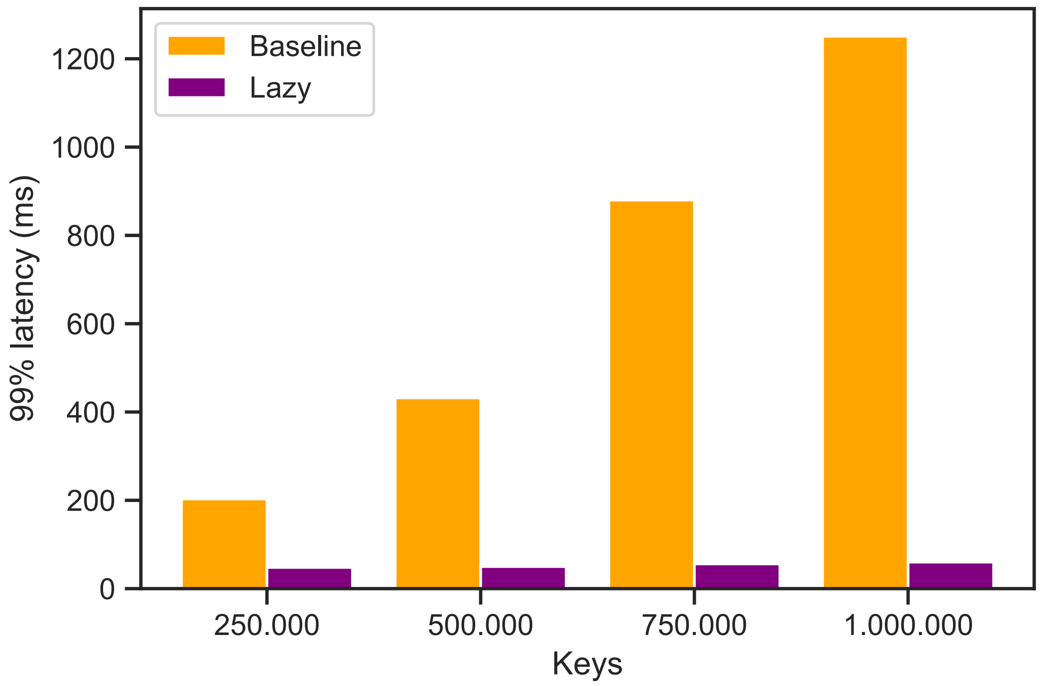


FIGURE 1.1: Peak latency comparison between baseline and the lazy approach, under various state sizes.

To that end, we aim to shed some light into the work being done on the matter, the ins and outs of the problem at hand as well as design an approach that tries to both satisfy the requirements of the underlying system and provide a non-blocking solution. An overview on the findings on the latter, can be observed in Figure 1.1, where an impressive performance improvement is evident. Most importantly, the performance benefits persist and even increase under progressively bigger loads.

Over the next chapters, we will analyze the background of the problem at hand, accompanied by the definitions and challenges it brings to the table, followed by presenting and commenting on existing solutions and academic endeavours suggesting different approaches. Furthermore, we will present an overview of our own approach, explain the reasoning behind the different design choices, as well as an implementation of this architecture on an existing system. Finally, we will present the findings of experimentation on this test system, highlight the importance and insights drawn from the results and discuss what this all means for future work and possible next steps.

Overall, the work presented aims to answer the following research questions:

RQ 1 : How is state migration defined as a process?

RQ 2 : What challenges come with designing a non-blocking solution?

RQ 3 : Can a lazy fetch approach offer a solution to the problem?

2 Background

In this chapter, we will try to define and outline all the different points involved in designing and analyzing solutions, on the topic of non-block state migration in stream processing engines. This will help the readers get closer to the technical aspects of the issue at hand, without requiring extensive background knowledge.

2.1 Definitions and terms

Stream Processing Engines/Systems (commonly referred to as SPE), are "big" data processing systems, designed to handle continuous processing of inputs characterized by velocity, variety, volume and veracity. The characterization of "big" (as opposed to "normal") data, carries with it the implied challenges of dealing with data that fall out of the norm and require specialized solutions in order for the end system to satisfy certain latency, cost and consistency restrictions. The streaming part comes from the fact that such systems process data "as they arrive", continuously producing results. Alongside streaming, batching enjoys a lot of the same advantages and disadvantages, following a similar design paradigm, but with processing taking place in batches (groups) of data. There, any guarantees offered by the system are applied on a per-batch basis, rather than a per-record basis.

Operators are the rudimentary processing units of a SPE and they apply certain transformations on user-defined functions on input data to produce intermediate results. As already mentioned in chapter 1, SPE dataflows can be modelled as a tree or a DAG (Directed Acyclic Graph) where data flows from the input into different operators, to potentially other operators connected to each other and passing intermediate result data from one to the next, until processing has concluded and the final analysis results are produced on the output. In essence, operators correspond to the different steps involved in the desired analysis taking place and their discretization into standalone processing units allows for system design flexibility.

Workers are the system-specific processes which are responsible for assigning resources to the calculations taking place. Assignments of operators and their replicas are made to available workers in a way specified by the system itself and create a link between the analysis carried out and the available resources. Worker assignments are important for efficient distributed execution and affect rescaling scenarios, but their allocation is a separate problem on its own and is also usually omitted.

Persistent data or states are called so when their discovery is disconnected from any runtime exceptions or general system failures. Essentially, it entails saving in a disk or other persistent medium and persistent information is deemed "safe". In fast-paced systems where throughput and latency are important aspects, processing happens almost entirely in-memory in favour of faster data access and transferring speeds. Literature, such as the work by Del Monte et al., 2020, often refers to the term "persistent" to highlight points in the design or process being presented, where a less volatile medium needs to be used, without going too deep into the technical details of how that happens. The latter is of course accepted to come with a potentially heavy latency/cost penalty.

State support on operators is extremely important, as it unlocks a slew of possibilities for operator tasks, beyond the simple map or add operation. Even an operation as simple as an average of a series of numbers requires the (temporary) saving of the sum and count of previously seen data records and (potentially) their respective keys. Together with the added capabilities however, stateful operators pose a number of challenges on the important requirements of the underlying SPEs as state needs to be considered on any consistency and fault-tolerance claims as well as specially treated during reconfiguration scenarios.

Partitioning happens on any data flowing through the system, usually in a key-based manner and ensures a deterministic path for each data record and aids in any consistency guarantees and fault-tolerance processes. Moreover, key usage statistics can be exploited to load-balance the system and offer overall greater throughput. The way key-worker allocations are made is the topic of research on its own and offers a lot of optimization ground, as previous work has shown that the way keys are allocated before and after rescaling happens, which affects the size of the collective state that needs to be transferred between workers, greatly affect the process of state migration itself. In their work, Gu et al., 2022 showed that going from a usual uniform repartition strategy to a simple consistent hashing one, yielded a reduction of over 85% on keys and states needed to move which translated into a subsequent 70% reduction in the rescaling duration and 90% reduction in the max latency observed over their benchmark. Even without looking into the specifics of the migration process itself, it is glaringly obvious that simply having less to transfer to achieve the same result, is faster and therefore preferred.

Consistency is a very sought-after property of data processing systems and whose concrete definition has been the topic of many a discussion and research. Early database systems defined the concept of ACID (Atomicity, Consistency, Isolation, Durability) to assert and guarantee data validity throughout the different operations being applied and transactions they are usually grouped into. In modern SPE literature, consistency guarantees are equaled with *exactly once* semantics, which according to Silvestre et al., 2021 means that *"an incoming record will apply its effects to the computation state of the system exactly once, even in the event of failures."*. The basis of such claims stems from the fact that distributed systems' execution is inherently hard to analyze and detail, partly due to their distributed nature and partly due to their fail-over capabilities. The only way to guarantee such restrictive properties is to try and show equality between the execution scheme of the distributed system and that of simple sequenced procedure that applies the same analysis operations one after the other.

Fault Tolerance is an integral part of SPE solutions, as systems that operate in a distributed manner require the ability to recover from partial failure on the underground infrastructure, whether that means that processes within machines or machines themselves stop working. Although at first glance the issue of fault-tolerance might seem independent to that of state migration, solutions such as Rhino presented by Del Monte et al., 2020 provided a holistic solution that handled both.

Reconfiguration provide an external functionality of the system, where resources, operators and workers can be re-allocated to match the load, satisfy new execution requirements or simply updates to the analysis process taking place. One reason why reconfigurations might happen, are for rescaling purposes, which means to introduce more resources to the system for improved performance or cut down on existing resources to mitigate costs. Scaling however is not always the goal and reconfigurations can also take place because the execution scheme has been updated or load balancing is needed to alleviate imbalances in the data assigning strategies throughout the system. The latter will, for the scope of this work, be referred to as reshuffling, in order to distinguish from the rescaling scenarios. Whether the case is reshuffling or rescaling however, stateful operators pose a challenge in consistency and overall system performance when reconfiguring, as taking care of this extra data leads to the problem at hand, known as state migration.

State Migration refers to the process of moving state among workers as a direct result of changing a number of operator instance - key mappings in the system, while being able to resume normal execution. To achieve such a task, there is a number of important sub-tasks that need to be orchestrated, executed and verified before normal execution can resume. Moreover, maintaining any consistency guarantees throughout the process, is especially tricky and the main reason why early solutions resorted to offline approaches where execution altogether is halted, the current state of the overall system is saved, new configuration submitted and state restored, before execution can be re-started. Such processes are of course slow and attribute a particularly high cost to any reconfiguration calls, effectively prohibiting frequent usage of the feature.

2.2 SPE Requirements

Designing and proposing a Stream Processing Engine architecture, carries with it a number of limitations, in the form of requirements that such systems typically need to satisfy. Stream processing engines serve as the backbone of real-time data processing applications, enabling the efficient processing of continuous data streams and as such any appropriate solution needs to account for and satisfy the following requirements.

2.2.1 Scalability and Elasticity

A robust stream processing engine should offer horizontal scalability, allowing it to handle increasing data volumes and processing demands. The engine should also support elasticity, which involves automatically adjusting resources based on workload fluctuations to maintain optimal performance. An equally important point is that such abilities be masked from the end user, effectively painting the picture of a system capable of any given workload.

2.2.2 Low Latency

Stream processing engines are primarily used to process data in real-time. Therefore, low latency is essential to ensure that data is processed and analyzed as quickly as possible, enabling timely insights and actions. Similarly to before, any internal actions taken by the system to adjust to changes, failures or scaling, should happen in a transparent way that does not affect performance, as much as possible.

2.2.3 Processing Guarantees

Stream processing engines should support various processing guarantees, such as at-least-once, exactly-once, or at-most-once processing semantics. These guarantees determine how data is processed and ensure data integrity in the face of failures. Effectively gauging the ability of a given system to provide processing guarantees is the topic of discussion and research, within the distributed systems community.

2.2.4 Fault Tolerance and Reliability

Given the distributed nature of stream processing systems, they should be resilient to hardware failures, network issues, and other potential disruptions. The engine should provide mechanisms for data recovery and fault tolerance to prevent data loss and maintain system reliability.

2.2.5 State Management

Many stream processing applications require maintaining state information over time, such as aggregations or session windows. The engine should provide efficient and scalable mechanisms for managing and updating this state while minimizing the impact on processing speed.

2.3 State Migration Motivation

Before any other important aspects of the issue can be explored, it is important to ask *why* it is really needed in the first place. As already mentioned in Chapter 1, the existence of state within the worker nodes, allows for more code flexibility and bestows upon the end system the ability for further and more complex calculations. As such, the need for a foolproof way of managing this state is apparent. Furthermore, the ability to transfer this state between worker nodes, known as state migration, further aids the whole system by enabling a series of features.

Fault Tolerance and Reliability: In distributed systems, hardware failures, network issues, and other disruptions are inevitable. State migration allows stream processing applications to recover from such failures without data loss. By transferring the application state to healthy nodes, the system can resume processing without interruption, maintaining data reliability and application continuity.

Load Balancing: Distributed stream processing applications often experience varying workloads across processing nodes. State migration enables load balancing by redistributing state across nodes based on their processing capacities. This prevents resource bottlenecks and ensures optimal utilization of system resources.

Scalability: As data volumes and processing demands increase, stream processing systems need to scale horizontally. State migration facilitates the adaptability of the system by allowing new processing nodes to join the cluster and receive the required state, or old ones to hand over their own state before leaving. This scalability ensures that the application can handle growing workloads effectively ,or conversly scale down in case of reduced workload to keep costs low.

Dynamic Resource Allocation: Stream processing engines often operate in dynamic and elastic environments, where the number of processing nodes may change frequently. State migration supports dynamic resource allocation by allowing nodes to be added or removed while ensuring that the necessary state is transferred to maintain processing continuity.

Seamless Code Updates: When stream processing applications are updated or upgraded, the state migration mechanism assists in transitioning to the new version without data loss. The migration process ensures that the state is compatible with the updated application logic, allowing for a smooth transition.

All of the above highlight the importance of state migration in enabling Stream Processing Engines to cover a plethora of crucial features, as well as ensuring others operate as intended. Performing the task in a seamless, cost-efficient and fast manner is equally important, driving research over the past years on a solution that achieves **online state migration**, without affecting execution or disrupting the dataflow throughout the system.

2.4 Online State Migration

Performing reconfigurations without affecting normal execution of the system, is an attractive feature that further facilitates system elasticity, performance and transparency but does not come without a slew of new problems and limitations.

2.4.1 Definition

To better understand the problem, a common basis of what constitutes a complete online state migration, must be established. A single reconfiguration, can result in multiple state migrations. For each of these migrations, a change of ownership for a set of record keys takes place where the previous (origin) worker hands over responsibility for the set of keys to a new (target) worker. The process as a whole, can be divided into the **route update** and **state transfer** steps or sub-tasks. With most systems, completing all necessary state migrations following a reconfiguration command typically consists of first updating all affected routing tables and then actually transferring the state between workers to facilitate normal operation afterwards.

With the **route update** step, any routing table entries responsible for routing records within the execution graph of the system are updated, to reflect changes resulting from the reconfiguration command. This means that affected records will now be routed to the target worker, instead of the origin worker.

Once routing has been sorted, the **state transfer** step can be performed. The state of the keys being migrated must be transferred, so the target worker can continue processing input for the keys being migrated seamlessly.

2.4.2 Truly "online"

Throughout this work, the term *online* will be used to refer to solutions seeking to perform the necessary steps towards system reconfiguration, without halting execution or generally disrupting the dataflow. However, it should be noted that as the necessary steps require some rudimentary processes of their own, currently proposed solutions are not truly "online", but rather provide an improvement over simple and completely blocking solutions. The discrepancy between the state migration completion time and the records arriving in the meantime, is handled by buffering mechanisms, which each system employs differently to attempt to mitigate the latency introduced to the system as a result. Perhaps a completely non-buffering solution would be called truly online, but falls out of the scope of our research and possibly the field at the moment.

2.4.3 Challenges

The previous clear distinction between the route update and state transfer steps, involved in an online state migration process, helps identify the key points and challenges of the task at hand.

Routing updates throughout the system, entail updating entries in routing tables contained in all nodes of the graph, which in turn help decide where each record should be routed to next, depending on its corresponding key. Because of the acyclic connected graph-like natures of SPE systems, one way such a task can be performed, without the need for any intermediate buffering of in-flight records, is by special "control message" records that traverse through the graph and update routing tables on-the-go. This in turn means that routing table updates happen asynchronously throughout the system, meaning that determining that records flowing through the system at the same time will encounter "old" or "updated" routing tables is impossible. Therefore, a better way of handling routing updates while records still flow through the system is required.

State transfers suffer from a similar limitation. Moving sets of data, especially of unknown size or structure, inherently carries with it a delay penalty. During that delay, records whose processing requires the very state being transferred can arrive, leading to latency being observed in the system. Regardless of whether micromanaging can occur to ensure not all affected states needs to be transferred before the in-flight records are process, an optimised way that ensures the required state for any arriving record can be used in a timely manner, is highly important.

Finally, synchronizing the two steps and executing them in tandem presents with a challenge of its own. A simple approach to attempt and preserve consistency throughout the process, would find a system execute all routing updates for all keys affected, before moving on to state transfers between origin and target workers. The main idea behind this decision, being that consistency can only be claimed when only one worker at a time holds and updates the state for a given key.

3 Related Work

Online state migration is a much sought-after feature on modern Stream Processing Engines and its importance has been highlighted already throughout this work. A number of different published works have attempted to deal with the problem, employing different techniques and architectures. The ideas behind these works, as well as their findings and insights on the matter, will be presented next.

3.1 Categorization

Earlier approaches on state migration mechanisms have previously been categorized by relevant literature (Hoffmann et al., 2019 Gu et al., 2022) into the following distinct categories.

1. **Stop-and-Restart:** A relatively "simple" approach, briefly mentioned in Section 1, sees the process begin with a complete halt of system executions. Next, the system-wide state is saved, the new configuration loaded and state restored before normal execution can be resumed. By relying on parts of the fault-tolerance mechanism already necessary for the system, such as saving the system's state and then being able to restore it, such approaches are able to make consistency guarantee claims. That is the main reason why widely-used systems, such as Spark Stream Kroß and Krcmar, 2016 or Apache Flink Katsifodimos and Schelter, 2016, belong in this category. However, such mechanisms do not come without their own shortcomings, as relying on persistent mediums leads to penalties in migration duration, system availability and overall latency.

2. **Partial pause-and-resume:** Reconfigurations often lead to changes in specific parts of the computation graph representing the SPE. Hence, approaches of this category seek to constrain any performance penalties only on the affected sub-graph, while allowing the rest of the system to continue functioning normally. Implementations such as SEEP Castro Fernandez et al., 2013 and Chi Mai et al., 2018 try to minimize the system availability penalty this way, but the technique can easily degrade into a stop-and-restart case, depending on the "importance" of the operators affected and the way they are connected to the rest of the system as noted by Gu et al., 2022.

3. **Dataflow Replication:** An important category of approaches, sees the use of replicas in the form of duplicate operators or sub-graphs, that can be simultaneously used to execute data processing tasks in both the new and old configurations, until the state migration process is concluded. While relevant system implementations, including Gloss Rajadurai et al., 2018 and Chronostream Wu and Tan, 2015, offer improved latency penalties during reconfigurations, they call for increased resource usage and special de-duplication techniques to cope with the replicated nature of the system consistently.

3.2 Megaphone

3.2.1 Design

Introduced by Hoffmann et al., 2019, Megaphone's main idea "augments" stateful operators with extra functionality that allows them to operate as both processing units as well as controlling units for the migration process. Figure 3.1 shows an overview of the novel operator design the approach presents, where (a) depicts a normal operator and (b) shows the new "augmented" operator and the extra parts it consists of. The design introduces two new operators denoted *F* and *S* to replace a traditional stateful operator denoted *L*.

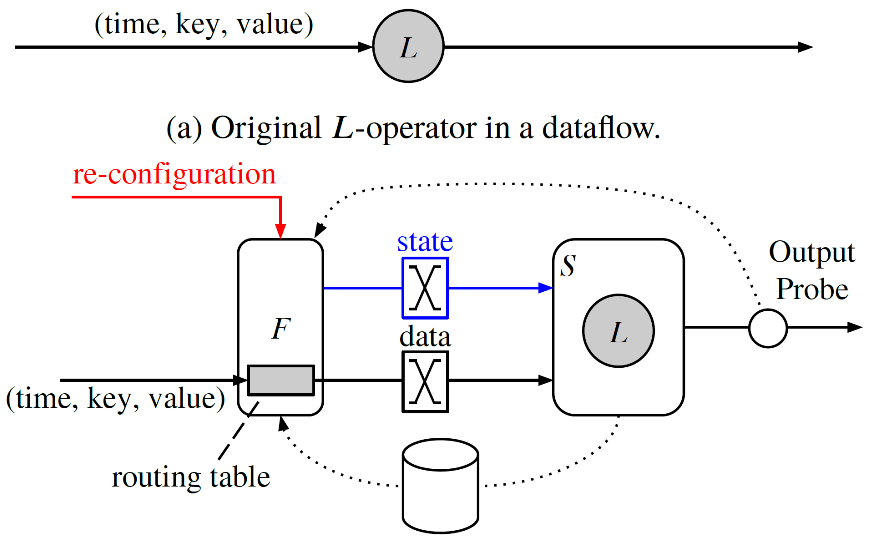


FIGURE 3.1: Overview of Megaphone's migration mechanism. Hoffmann et al., 2019

F operates as a secondary router for data records, since after they are routed to the enclosing- or L-operator, the mechanism requires the added ability to re-route required information to an operator of choice. Interestingly enough, *F* does not only bear the responsibility of further routing data records, but also migration control messages as well as state. This means that *F* is essentially responsible for initiating migration steps on connected peers, as well as providing the state and records that a stateful operator needs to operate on.

S on the other hand, takes the place of the initial L-operator as it receives records and processes them according to the stateful process it supports. However, the state is not retained locally by *S* itself, but rather provided and handled by its upstream *F* operator, while access is shared by both. In essence, *S* can be described as a bare processing unit, responsible only for applying updates on the provided state using the provided data, that does not in any way meddle with the migration process.

To handle the synchronization issue between routing updates and state transfers in a consistent manner, as explained in subsection 2.4.3, Megaphone employs a timestamp-based mechanism which assigns a *time* value to any data record or control message passing through the system's dataflow. This timestamp, can then be used to determine data records flowing through the system before and after a re-configuration signal has been issued. Therefore, processing of post-configuration records arriving **before** the reconfiguration has actually taken place, can be buffered until said re-configuration is completed and any resulting updates from the data records can take place. At any given time, the parallel operators maintain a "frontier" in the form of the last known timestamp for which both data records and potential reconfiguration control messages have been consumed, as well as those processed and presented to output. Frontiers allow the operators to assert that no records containing earlier timestamps will be introduced later on and any buffered records can resume to flow or be processed. Local F and S operators can both "peek" into each others' frontier, to accordingly decide if they should progress or not.

Perhaps interestingly, the approach calls for a mesh communication design between operators, denoted by "X" in the figure, where operators can route data to other operators, regardless of the underlying topology. Essentially allowing all operators to "talk" to each other unobstructed, regardless of whether in fact they belong to the same worker or even machine.

Given all of the above, the online state migration process with Megaphone becomes as easy as simply re-routing appropriate data records and required state from the origin F operator to the target S operator and resuming normal operation. An example scenario is presented in Figure 3.2, where three different steps are depicted that involve a state migration taking place, involving operators 1 (F_0, S_0) and 2 (F_1, S_1):

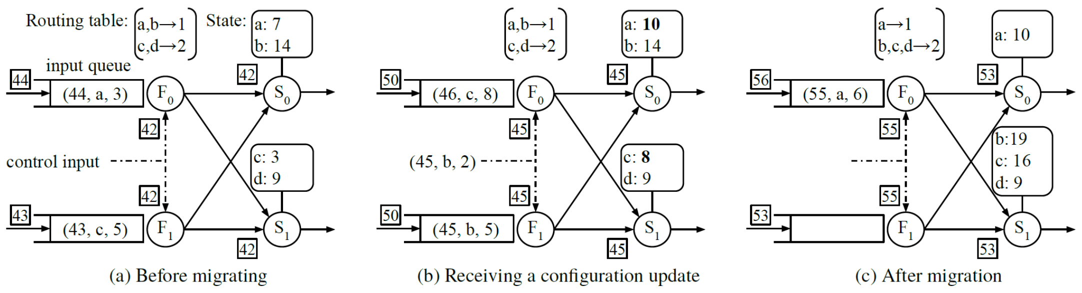


FIGURE 3.2: Example of state migration executed between two operators. Hoffmann et al., 2019

(a) All operators have progressed to timestamp 42 and are receiving records with later timestamps normally. The internal routing tables that both F_0 and F_1 retain locally are depicted, as well as the local states for the corresponding keys and their current respective values (although the figure depicts that state as local to S operators, we have already established that in fact access is shared between the two and state is in fact local to the overall operator). We can see F_0 receiving a record for key a that should be routed to S_0 and F_1 receiving a record for key c that should be routed to S_1.

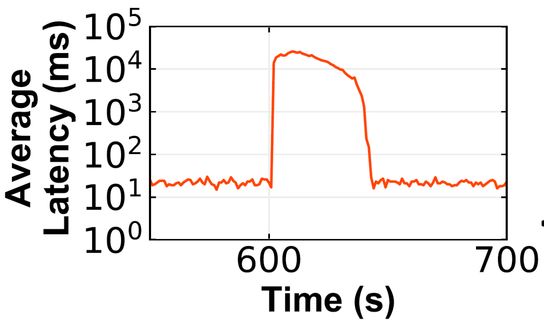
(b) After the records of the previous step are routed and processed normally, the states are updated accordingly and a control message calling for a reconfiguration arrives. This control message translates into "*operator 2 will be responsible for records of key b as of time 45*". In order to stay true to the command, the system needs to make sure that any records bearing key b and time 45 or greater, should be routed to and processed by operator 2. Therefore, when the frontier progresses past 45, F_0 needs to move the required part of the state to S_1, while both F_0 and F_1 need to start routing all records with key b to S_1. At the same time, we can see a message arriving for time 45, containing the key b at operator 2.

(c) Following a successful state migration and resuming normal operation, the appropriate "snapshot" of the system is presented. State for key b is upgraded and now residing within operator 2, as the frontiers have progressed past 45, following the previous reconfiguration message. The synchronizing nature of the timestamps and frontiers is also presented, as *S* operators have processed inputs with max times of 53 and subsequently observed records with time 56 are buffered. Once any control messages queued for the intermediate times arrive and local frontiers update past 56, the buffered record will be consumed.

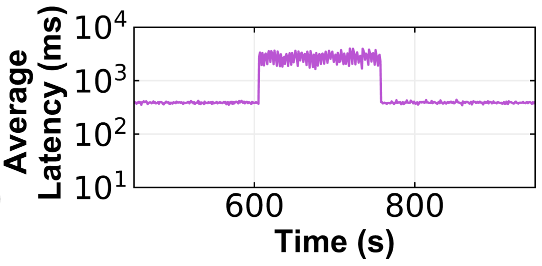
3.2.2 Advantages

The obvious benefit of the architectural choices Megaphone makes, stems from the fact that the process of state migration is incorporated within normal operation of the system and does not require special handling or new processes to run. By "augmenting" ready-state processing on operators, the system enjoys reduced performance hits during state migration phases.

Independent works also attest to the benefits, with Del Monte et al., 2020 only noting Megaphone's inability to handle bigger-than-memory states, which in their own words could easily be solved by "*the introduction of a data structure that supports out-of-core storage, e.g., a key value store (KVS)*". Furthermore, benchmarks performed by Gu et al., 2022 on a simple key-count example, show how Megaphone improves on the penalty introduced when going from a ready-state phase to a state migration phase. Specifically, Figure 3.3 shows how Flink, a frequently used baseline, "jumps" from 10^1 ms to 10^4 ms during reconfiguration, while Megaphone only goes from 10^2 ms to 10^3 ms.



(A) Native Flink



(B) Megaphone on Flink

FIGURE 3.3: Comparison of end-to-end latency on a key-count example between Flink and Megaphone. Reconfiguration at 600s. Gu et al., 2022

3.2.3 Disadvantages

On the other hand, the design choices Megaphone makes, come with certain weak points. The first and perhaps most important limitation can be identified in Figures **??** and **??** as the "ready-state" phase of the execution, when no migration costs drive the latency up, seems to be significantly hindered in the case of Megaphone which displays an average latency of almost 10^3 s., compared to Flink's 10 s. The increased latency comes as a direct result of the extra partition overhead that the inner routing and sub-operators' operation incurs. Even though the presented numbers come from a naive implementation of Megaphone on Flink, which allows for head-to-head comparisons, the outcome is to be expected as Megaphone sets a number of requirements, which are not natively supported by mainstream SPEs. These include the ability to extract state from upstream operators, as happens when splitting the original L operator into F and S, as well as the extra synchronization needed due to the timestamp-based dataflow frontiers. This essentially leads to increased latency on the system, which "violates" one of the basic requirements set back in Section 2.2.2.

Another potential drawback of the system, roots back to the ability of F operators to freely route data to all other S operators. In order to enable such communication, a mesh network between operators and therefore workers and physical machines needs to be established. As a result, a system implementing such a design does not follow the shared-nothing architecture, typically aiding with elasticity in distributed systems, as noted by Gulisano et al., 2012. The added cost of ensuring consistent mesh data communication channels between all involved parties, is not taking into account during evaluation, but simply noted as a prerequisite for setting up a similar system.

Finally, being mainly a downside of the associated research paper rather than the system design itself, Hoffmann et al., 2019 do not in any way facilitate or analyze the repercussions on fault tolerance. They merely note that Megaphone follows the timely dataflow abstractions and thus simple snapshot-based approaches could be built, failing to take a closer look into potential problems that the underlying design could introduce. For instance, how misaligned routing tables between peer F operators would be handled or partial failures during state migrations could be dealt with efficiently.

3.2.4 State Transfer Strategy

A secondary contribution made by Hoffmann et al., 2019, takes a look into how different state transfer strategies affect the behaviour of the system overall. After deciding to move state associated with a preset number of keys, following a reconfiguration, Megaphone's team argues there are three notable ways in which the associated state could be moved, all offering different trade-offs on different metrics of the system:

- **all-at-once:** where, as the name suggests, the entire state in question is marked for transfer and normal execution on the new operator can only be resumed after all of it has been successfully transferred.
- **batched:** splits the the keyed state in batches of keys and then moves the different groups one by one, until all of them are transferred.
- **fluid:** simply moves state on a per-key basis, effectively choosing one key at a time to be transferred, until all are completed.

The paper finds that there is a peek latency over system throughput improvement to be made when the latter two over the more "traditional" *all-at-once* strategy, as can be seen in Figure 3.4.

Even though moving all the state at once means that after the process is done, normal execution on all migrated keys can be continued uninterrupted, *batched* and *fluid* strategies are found to offer reduced overhead and communication costs while migration is under way. Additional experiments showed that the two "new" strategies lead to reduced max latency for different key bin sizes, while also exhibiting similar state migration duration as *all-at-once*.

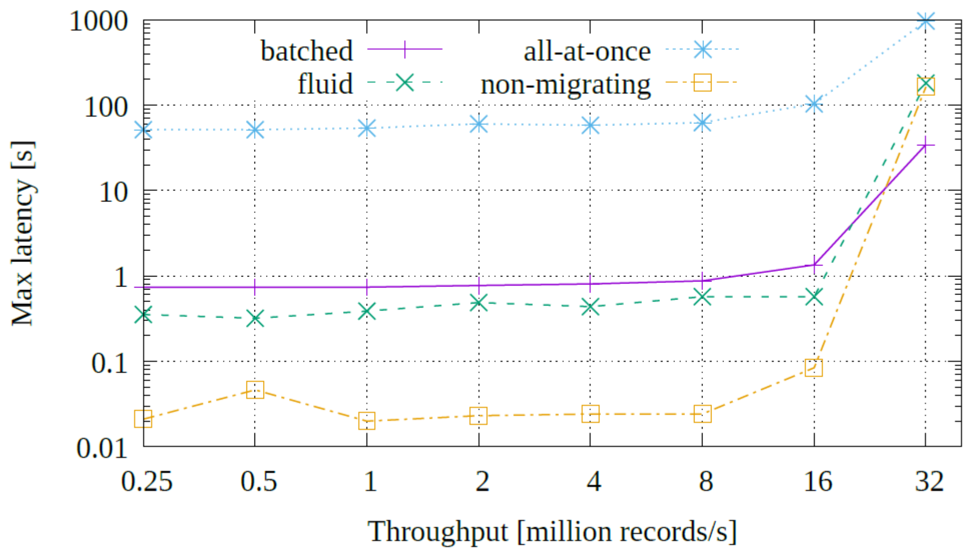


FIGURE 3.4: Offered load versus max latency for different migration strategies for key-count. Hoffmann et al., 2019

Perhaps most interestingly, this particular contribution sheds some light into the the ability to adjust migration costs and performance dynamically, by selecting the best bin size in the aforementioned state transfer strategies, to match the specific execution case, such as state size, number of keys or similar.

3.3 Rhino

Rhino is a novel approach, presented by Del Monte et al., 2020 which seeks to solve reconfiguration problems in the face of very large distributed state. Rhino employs what is also called as a "checkpoint-assisted" solution from Volnes, Plagemann, and Goebel, 2022, that utilizes snapshots for the migration process itself rather than just employing them to deal with fault tolerance.

3.3.1 Design

Rhino essentially devises an optimized distributed incremental checkpointing structure, that in turn allows it to retain updated snapshots of state potentially needed in the future, before the need arises. The idea behind such a structure is that if backups are local and up-to-date before a worker or its operators ask for state contained in them, then a fast local restore of that state could play the role of migration without the need to bear communication costs and delays at the critical moment.

A great tool that rhino puts to use to somewhat minimize routing costs on checkpointing, is consistent hashing on virtual nodes.

The system comprises of four distinct components, that deal with different important aspects of the system.

- **Replication Manager** that creates all the replica groups before assigning them to workers.
- **Handover Manager** that deals with all the synchronization issues of handling in-flight records while performing state migrations, as the problem was described in sub-Section 2.4.3

- **Distributed Runtime** is the main runtime code that runs on all workers and contains the necessary protocols for handovers and state-centric replications.
- **Modifications** in the form of stateful operator extensions necessary to implement the design. First, control events through the data channels are necessary for delegating control requests to appropriate workers consistently. Next, the state migration process itself requires buffering and inbound/outbound data channel re-wiring capabilities. Finally, the ability to consume the checkpointed state in the worker, is crucial to the overall success.

The way Rhino operates during a reconfiguration process, is presented on Figure 3.5, where an example scenario is shown. The steps shown, are described below and start with the Handover Manager inserting special "handover markers" into the dataflow, which traverse through the system, until they have passed from all operators :

Step 1 The markers are emitted from upstream (previous) operators to all connected downstream operators, to signify a state migration process occurring.

Step 2 At this stage, some markers from upstream peers have reached the intended operators, but some are still in transit. As soon as any operator starts receiving markers, a buffering on that channel is initiated, until all markers are received and a decision on whether migration also involves the operator in question can be made. For example, operator I will not actually play a role in the impending migration, but as long as both S_1 and S_2 have not confirmed so, no decision to continue normal operation can be made safely (since each is responsible for different subset of keys).

Step 3 All markers have reached their intended operators and a migration from O(rigin) to T(arget) is necessary. I turns out to not be involved and its input data channels can once again be consumed normally. T can use the local replica of the migrated state to gain the state associated with the affected keys, while O needs to re-wire its output in order to send all in-flight records on affected keys (that reached O during buffering) to T.

Step 4 After the state has been "transferred" to the target operator, the temporary data channel can be removed, input data channels resumed and handover markers passed downstream. Upon receiving all the markers, downstream operators together with O and T, can notify the Handover Manager of completion of the state migration process.

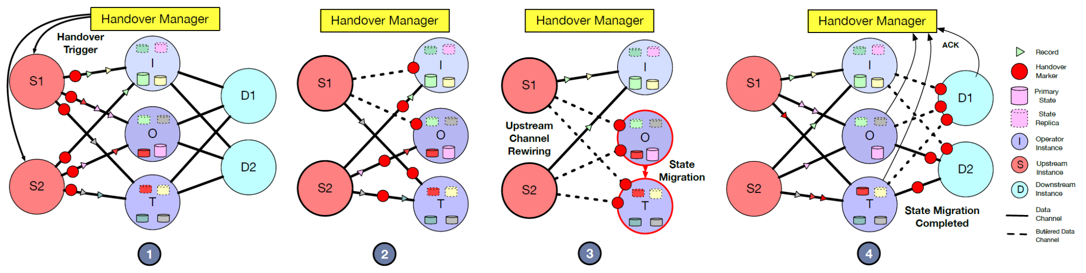


FIGURE 3.5: Comparison of end-to-end latency on a key-count example between Rhino and Flink. Reconfiguration at 600s. Del Monte et al., 2020

Once again, we are presented with a novel way to circumvent the routing update challenges described in sub-Section 2.4.3. We can see how the handover markers are used to assert the consistency and completion of the routing updates, while the internal re-wiring of the origin worker's output channels deals with the synchronization issue. This way, the routing updates are guaranteed to complete while in the meantime any data records routed inconsistently by peers due to the asynchronous nature of the updates, can be re-routed after the fact.

Another important aspect of Rhino, as evident from the look on the migration example, is its checkpointing solution that doubles as a state transfer solution among workers. The produced snapshots are then used as direct replicas of peer state. In order to make a checkpoint-assisted approach feasible, Rhino presents and evaluates a pre-emptive, distributed, incremental and replicated checkpointing structure. This means that the snapshots taken by the system, take place in a distributed manner where the checkpointed state is broadcasted to a group of selected peer workers and replicated among them. The checkpointing process itself is incremental, meaning only the changes on the data since the last checkpoint are stored, effectively minimizing new objects' size to save and communicate to peers. As the characterization "checkpoint-assisted" has already hinted about the solution, Rhino makes use of the aforementioned local replicas stored pre-emptively in the worker, to gain direct access to a copy of peer state. Snapshots typically reside within persistent storage and relying on persistent mediums for performance-oriented processes might prove cumbersome as they are notorious for being orders of magnitude slower than physical memory. To assist with this issue, Rhino changes the typical block-centric approach on snapshot replication to a state-centric one, so that it can ensure complete states reside within workers and do not need to be pieced-together from multiple peers. The last point, Del Monte et al., 2020 stresses is a major consideration, as their independent testing on Flink[1] and Megaphone showed that during recovery, most of the time is spent in state materialization from a previous checkpoint. As recovery in the case of Rhino is also used for state migration, it offers significant improvement.

3.3.2 Advantages

Rhino carries a completely different system design and as such offers grounds for discussion on pros and cons in almost every important aspect of a modern SPE. Starting off with the positive points, the proactive checkpointing strategy that sits in the center of the design, allows for "simpler" fault tolerance. By having restore points readily available on workers at any given time, Rhino allows for full restores in case of failures, without the need for extra fault tolerance services and handling. Backups with own state are local, up-to-date and can be directly used as restore points. Moreover, the state-centric approach aids in the restore process having to fetch data from less source and thus significantly speeds up the state materialization task. The benefits are presented in Table 3.1, where an example scenario of recovery following a vm failure is used to compare Flink, Megaphone, Rhino and RhinoDFS. The last is a variant of Rhino employing a DFS instead of Rhino's elaborate consistent hashing structure and highlights block-centric vs state-centric replication.

Perhaps the main strong suit of Rhino, is its focus on "big" state. By relying on persistent mediums for state transfers and the ability to fetch state from them,

[1]Apache Flink: `https://flink.apache.org/`

State Size	System	Scheduling	State Fetching	State Loading
250 GB	Flink	2.2 ± 0.1	68.2 ± 5.7	1.3 ± 0.2
	Rhino	2.8 ± 0.2	$\mathbf{0.2 \pm 0.1}$	1.3 ± 0.3
	RhinoDFS	2.9 ± 0.2	10.7 ± 3.1	1.3 ± 0.3
	Megaphone	46.3 ± 2.8		
500 GB	Flink	2.5 ± 0.2	116.6 ± 4.9	1.8 ± 0.3
	Rhino	3.1 ± 0.3	$\mathbf{0.2 \pm 0.1}$	1.3 ± 0.3
	RhinoDFS	3.0 ± 0.3	18.9 ± 3.7	1.3 ± 0.5
	Megaphone	74.8 ± 3.0		
750 GB	Flink	2.6 ± 0.3	205.3 ± 5.2	1.3 ± 0.1
	Rhino	3.0 ± 0.2	$\mathbf{0.2 \pm 0.1}$	1.5 ± 0.1
	RhinoDFS	2.6 ± 0.1	36.1 ± 2.3	1.5 ± 0.2
	Megaphone	Out-of-Memory		
1000 GB	Flink	2.4 ± 0.3	252.9 ± 5.9	1.5 ± 0.2
	Rhino	3.0 ± 0.2	$\mathbf{0.2 \pm 0.1}$	1.5 ± 0.2
	RhinoDFS	2.9 ± 0.3	62.7 ± 0.9	1.5 ± 0.1
	Megaphone	Out-of-Memory		

TABLE 3.1: Duration of state migration during a recovery, broken down into stages involved. Del Monte et al., 2020

it can support terrabyte-sized state. In fact, in solutions like Megaphone that use networking to facilitate inter-operator communications in parallel instances of different workers, the cost associated with packing, un-packing and transferring such big state could be equal or even worse than using elaborate persistent medium solutions. That means that in the realm of "big" states, the approach Rhino takes might be optimal in terms of performance as well.

3.3.3 Disadvantages

In order for Rhino to ensure as much as possible up-to-date replicas of state on peer workers, a high cost in the form of network bandwidth must be paid. Del Monte et al., 2020 themselves report a 30% usage during a replication, regardless of any other tasks being perform by the system at the time. In fact, Gu et al., 2022 confirmed this number, but expanded by selecting different replication intervals. Going from the default 10-minute interval to a 3-minute one saw an increased bandwidth usage of 46%, while the same number grew to 56% when selecting a 1-minute replication interval. Perhaps most importantly, this overhead is associated with a non-rescaling or steady-state processing case, meaning that Rhino increases cost throughout normal operation to account for special use-cases such as reconfigurations and fault recovery.

The design choices Rhino makes, have a negative impact on reconfigurations and specifically the case of scaling out by adding more workers to the system. Since no local checkpoints exist in these new workers, a global state migration must take place, where the entire checkpointed state is transfered, rather than smaller incremental parts. This incremental transfer of state is crucial to the system's performance, otherwise the system's state migration performance can degrade to a partial-pause approach, exhibiting latency spikes.

In use-cases such as the queries found in the frequently-used NexMark benchmark by Tucker et al., 2008, recurring read-modify-write operations are present. Coupled with the trade-off in selecting replication intervals, these operations can create a long tail of changes between checkpoints, further hindering the performance of transfering-what the system considers to be-small batches of changes between workers.

3.3.4 Pre-emptive state transfers

The novel design approach Rhino makes, aiming to improve performance of both state migration and fault tolerance in one fell swoop, introduces an interesting idea, regarding the admittedly costly task of transfering information between workers. By pre-emptively moving state between workers, it effectively disconnects the cost associated from the command that would otherwise initiate a transfer, such as state migration or fault tolerance. This way, a more spread-out resource usage is observed, especially when compared to performing the task as a direct result of the associated commands. The differentiated view on cause and effect around the tasks involved with state migration, was a major inspiration for the work presented in Section 4.

3.4 Meces

Gu et al., 2022 introduce an innovative system design called Meces, in an effort to improve upon all the previous takes on online state migration. By focusing on what they call "order-aware" migration, they hope to better serve in-flight records during migration and improve overall performance, without any extra steady-state costs incurred.

3.4.1 Design

The idea of dealing with the order keyed states are migrated consists the driving force behind Meces' design and the motivation behind it is illustrated in Figure 3.6.

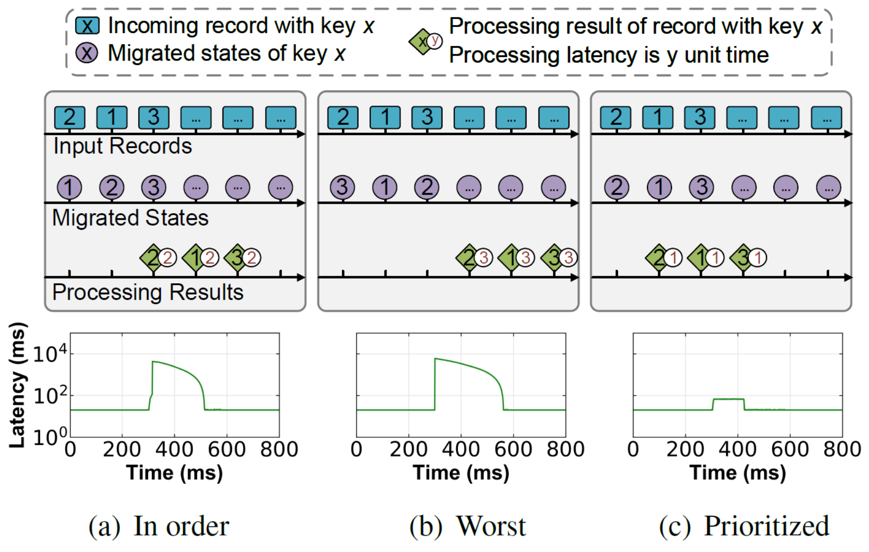


FIGURE 3.6: Observed event latency depending on migration order used. Gu et al., 2022

There, we can see a simple scenario depicted where states for 3 keys are being migrated, while records for the same keys arrive for processing by the system in question. The order the records arrive is fixed, but the order the states needed arrive on the target worker, is examined as:

(a) "In Order": The keyed states arrive in a random order, different to that of they keys of the records.

(b) "Worst": The keyed states arrive in the exact opposite order as the keys of the records.

(c) "Prioritized": The keyed states arrive in the same order as the keys of the records.

Even though at first the whole approach might seem unnecessary at first, the results suggest otherwise. Not only does the observed latency-the time the record spend in the system after being presented in the input and before arriving at output-in the prioritized case peak two orders of magnitude (from $10sec$ to $100ms$) lower compared to both the other approaches, but that increase in latency lasts for significantly less time (from $300ms$ to $110ms$). This essentially means that the system, incorporating order into state migration, can perform migrations faster, while attaining better performance on the processing of records arriving during the migration process.

To implement such an architecture and orchestrate migrations efficiently and consistently, Meces uses a global controller to initiate the migration process. The controller injects special control messages into the source operators and every operator I receiving the control message in any of its input channels, follows the following steps:

1. I sends the control message to all of its downstream operators.
2. If any of I's downstream operators will migrate states, I needs to update its routing table.
3. If I needs to migrate its state, it goes into its *aligning* phase and will later go into its *aligned* phase.
4. It notifies the global controller, who in turns awaits notification from all operators before the migration process can be deemed completed.

As mentioned, there are two noteworthy phases for operators in the process, namely *aligning* and *aligned*. An operator enters into the *aligning* phase once it has received at least one control message from its upstream operators, but still not all of them. After all upstream have successfully sent their control messages, the operator enters the *aligned* phase. The reason why this distinction is important, is because in the first case, **some** routing tables necessary have been updated but not all of them. As a direct result, records with keys whose state is in the process of being migrated from a *source* to a *target* operator, can still end up in either or both operators. Only after reaching the *aligned* phase, can an operator be certain that all keys will be routed correctly and it will receive only those it is responsible for.

Previous works, such as Rhino by Del Monte et al., 2020, have not observed this in-between phase and would only handle the period after routing updates have started and until they have concluded, by buffering arriving records. Meces in contrast, introduces its "order-aware" novelty at this point and allows operators in the *aligning* phase, when met with records whose state is not local, to fetch the associated state from the operator that actually holds it. Thus, records arriving in the meantime can rely on this fetch-on-demand feature to avoid having to wait for the entire process to conclude before they can be processed.

Following all of the above and after the operator has successfully entered its *aligned* phase, states can safely be moved to their intended target operators. For both the fetch-on-demand and the subsequent state transfers at the end of the state migration stage, Meces employs an external KVS (Key-Value Store). Specifically, Redis[2] is used to allow state transfers between parallel operators that otherwise share no direct communication channels, without mesh communication networks such as those necessary in Megaphone by Hoffmann et al., 2019.

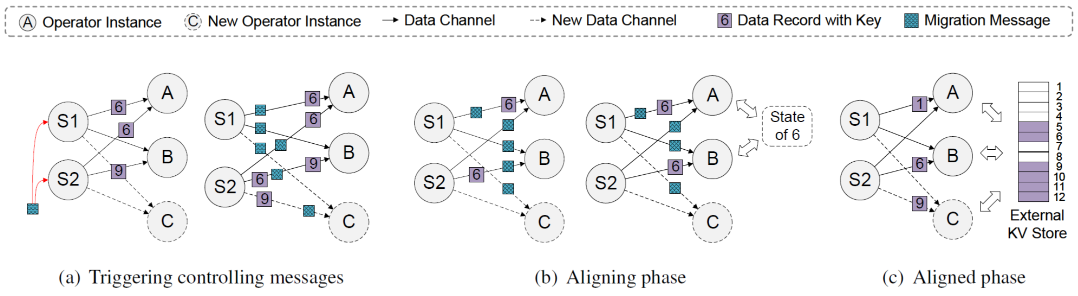


FIGURE 3.7: Overview of state migration process in Meces. Gu et al., 2022

An illustration of the process described above, is available in Figure 3.7. There, we can see the global controller injecting control messages to start the state migration process in (a) and records for key "6", which is being migrated from operator A (source) to B (target), being routed to A before the control messages and the subsequent routing table updates and to B after them. The newly exploited phase can be seen in (b), where although $S1$ routes "6" to A and $S2$ routes it to B, the fetch-on-demand feature allows normal processing regardless of who holds the state at any given time, even if that potentially creates a "ping-pong" case where the state keeps bouncing back and forth between two operators, before all upstreams are updated. Finally, after the operators are aligned in (c), a state transfer can be initiated to move the corresponding states to their intended targets.

Dwelving deeper into the granularity of the key-groups contained within state migration instructions, Meces borrows a page out of Megaphone's book and examines the case of "gradual" state migration. In essence, instead of initiating a migration on the initial request set of key-groups, Meces does migrations in subsets of the original set, until all the initial keys are moved. Figure 3.8 shows how that is done in Meces, in a simple scaling scenario from 2 workers to 3, where the keys are evenly re-distributed.

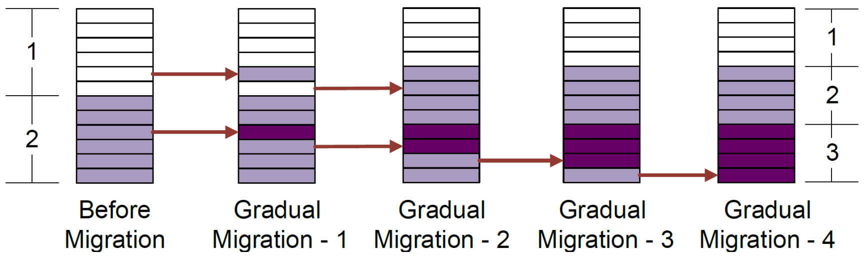


FIGURE 3.8: Meces gradual migration. Gu et al., 2022

[2]Redis Sentinel: `https://redis.io/docs/management/sentinel/`

The authors state that a gradual migration technique is employed to avoid latency peaks and achieve an overall smoother state migration process. In our understanding, the benefits of a gradual migration stem mainly from having smaller batches of transfers at the end of the *aligned* phase, as well as overall having more *aligning* phases, enjoying increased benefits from the fetch-on-demand functionality.

3.4.2 Advantages

The potential benefits of this "order-aware" approach have already been peeked at in Figure 3.6, but more extensive testing factoring in the design presented in Section 3.4.1, shows impressive performance improvements versus both Flink and a simple block-based "order-unaware" solution. Figure 3.9 presents the results of running a number of different tasks and then prompting a reconfiguration at 600 seconds. In all cases Meces outperforms, not exhibiting the immense latency spikes of 3 to 4 orders of magnitude higher than normal operation that Flink and sometimes Order-Unaware does. Moreover any observed disturbance to the system's processing performance is both minimal and short-lived, allowing the system to operate almost uninterruptibly and with high throughput, during the task at hand.

Being the most recent out of the approaches presented so far in this work, Meces aims to not only present a completely new way of operation for SPEs, but also improve upon the weak points that approaches like Megaphone and Rhino exhibited. Namely, a need is formulated by the authors to bring improvements by avoiding to introduce extra steady-state costs, as well as no extra resource overhead. In fact, the paper by Gu et al., 2022 performs direct comparisons to both systems, implemented on top of Flink based on their papers, in a similar case as before. An example processing task is handed and then a reconfiguration is triggered at the 600TH second of execution. The findings are shown below.

In the case of Figure 3.10, Meces can be seen operating on low latency an exhibiting minimal spikes during reconfiguration for a duration of 10-20 seconds. Megaphone on the other hand, incurs added partition overhead as discussed in Section 3.2.3, leading to a steady-state latency 10 times higher. The spikes during reconfiguration are proportionally similar to Meces, but since the steady-state latency was already much higher, they are also much higher than the ones Meces showcased.

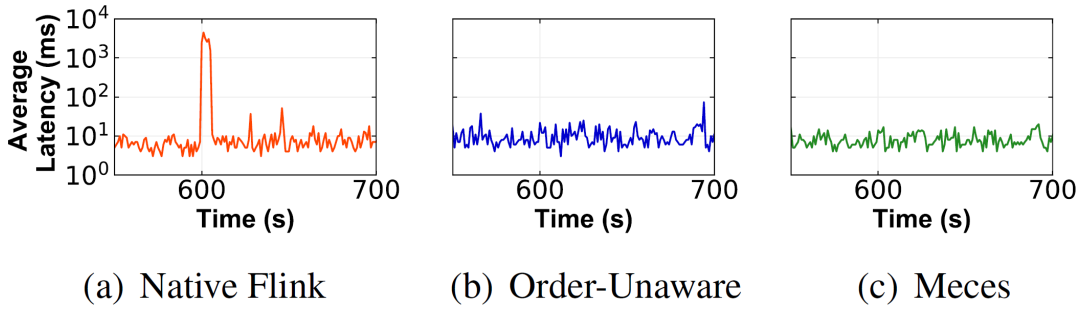
In Figure 3.11, a comparison with Rhino is presented. Since the sample task contains a scale-out case of reconfiguration, Rhino's relevant shortcoming discussed in Section 3.3.3 comes into play, causing it to show a high latency spike during reconfiguration. Since a global checkpoint transfer is required to take place, the performance observed is comparable and almost similar, to that of the Order-Unaware solution presented before. At the same time, Rhino incurs a high network bandwidth cost to operate, unlike Meces.

On the consistency front, Meces claims to not present any new threats and as for the novel fetch-on-demand feature, since only a single operator is responsible for the state of one key at any given time, updates are consistent and so should results be.

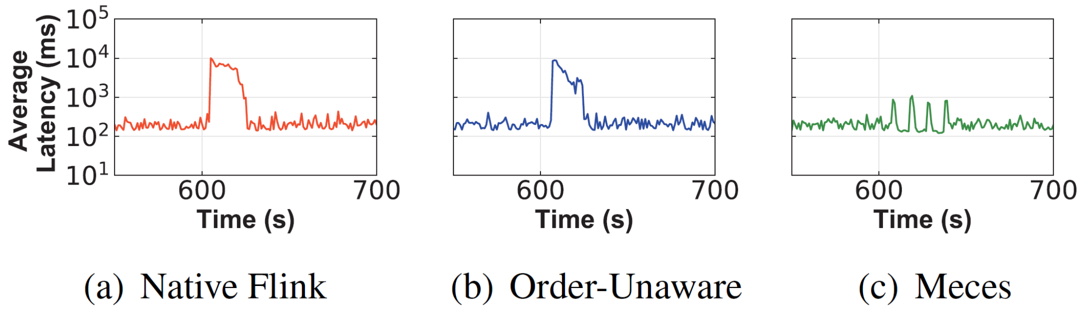
3.4.3 Disadvantages

Meces' novelty essentially finetunes migration steps to the order records arrive, making sure any necessary state is present at the correct place at the correct time. The fetch-on-demand capability helps achieve that during the somewhat unstable *aligning* phase, but not without a caveat. The "ping-pong" case briefly mentioned in Section 3.4.1, is actually noted as well by the authors themselves, although they dismiss it as seemingly meaningless, based on the fact that the *aligning* phase is relatively short-lived. However, this claim carries with it a potential limitation on the benefits the system enjoys as a direct result of the "order-aware" approach, making for a key observation that led to the design presented in Section 4. The gradual migration trick somewhat mitigates the issue, but does not directly address it.

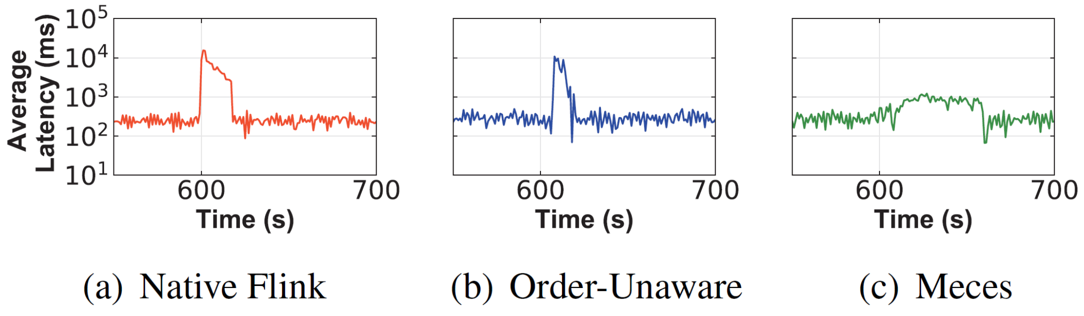
On the fault tolerance front, Meces makes no contributions, relying on existing fault-tolerance through checkpoints. However, the extra handling in the *aligning* phase Meces performs, requires special care and as such the authors advise pausing checkpointing for the duration of the state migration process. Such a limitation is not very welcomed, as fault tolerance plays a crucial role in solidifying SPEs as stable and complete processing solutions. Even if a case could be made about the fact that migrations are relatively short-lived, a potential solution such as that presented by Rhino where incremental checkpoints are employed to significantly speed-up the process of creating restore points, a halt on the process would significantly increase the list of changes and the incremental checkpoint's size.



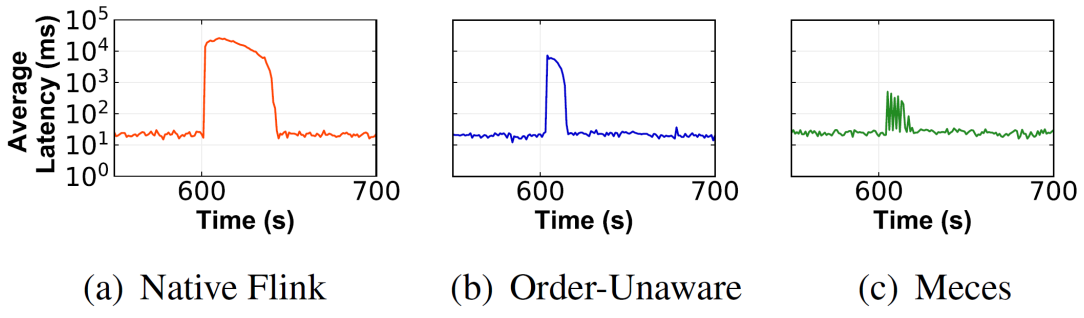
(A) NEXMark Q1



(B) NEXMark Q7

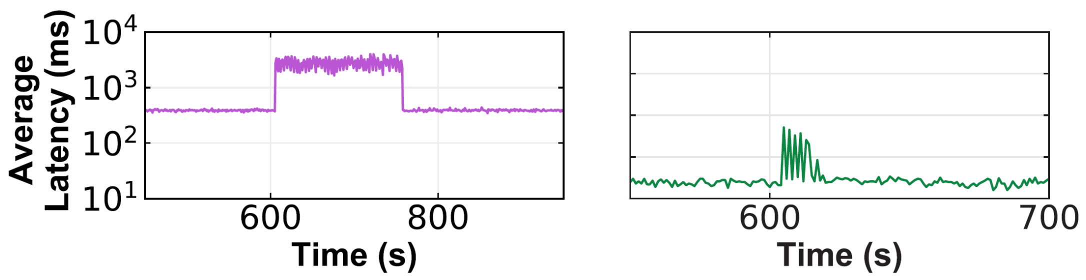


(C) NEXMark Q8



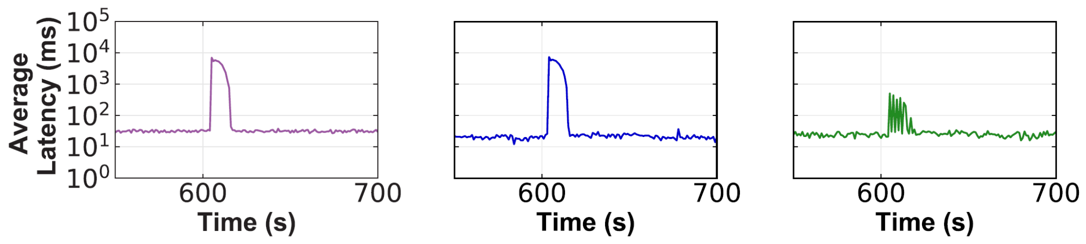
(D) key-count

FIGURE 3.9: Observed end-to-end latency. Reconfiguration triggered at 600s. Gu et al., 2022



(a) Megaphone on Flink (b) Meces

FIGURE 3.10: Latency comparison Meces vs Megaphone. Reconfiguration triggered at 600s.Gu et al., 2022



(a) Rhino on Flink (b) Order-Unaware (c) Meces

FIGURE 3.11: Latency comparison Meces vs Rhino. Reconfiguration triggered at 600s.Gu et al., 2022

4 Methodology

The research presented so far in the present work, has proven that designing an efficient online state migration solution that abides by requirements and good practices, is extremely hard. Consistency, fault tolerance, efficiency and processing latency seem to be almost mutually exclusive, with most approaches having to sacrifice one to benefit another. Below, we present and go over the details of a "lazy" approach on online state migration, in the form of an architectural decision, before we analyze the repercussions that such a design choice has on important aspects of modern SPEs.

4.1 Novelty

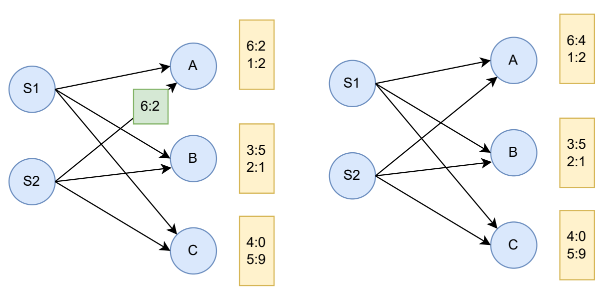
The root of the idea behind the lazy fetch approach, is inspired by all of the work presented throughout Section 3. The first important point being the interesting view Rhino presents on the cause and effect relationship between signaling a migration and initiating the costly work to support the operation. The other important inspiration being the fetch-on-demand approach by Meces and the entire analysis on the benefits of "order-aware" or simply put timely availability of state in operators.

To that end, we can observe a pattern where all approaches follow a somewhat similar strict paradigm on the phases of a state migration process:

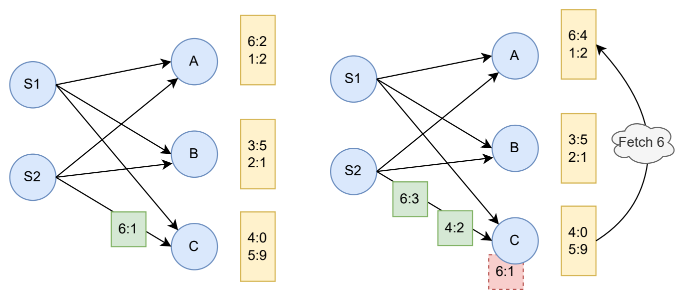
1. *Signaling*: Through either markers, special control messages or external controllers, operators are instructed to start preparing for a state migration.
2. *Syncing*: Some updates, usually on routing tables, have taken place but not all are complete. Therefore the system is in an intermediate phase which is not truly stable in terms of processing consistency yet.
3. *Transfer*: The updates have taken place, new records will be routed according to new configuration and any state not moved yet, can now be safely transferred to their intended destination.

Especially the latter two, *Syncing* and *Transfer*, contain performance traps as buffering is usually employed to deal with moments where state is not available as new records arrive. Either due to network communication latency or simply too much data to move, records are left waiting for the state to become available and processing to continue taking place. Inevitably, the discussion about this new approach turned into a question about what would it take to completely remove these two phases. Since a fetch-on-demand implementation can ensure state is present on an operator when needed, even in the "dangerous" *Syncing* phase and the cost associated with moving **all** state after the phase is completed is quite high, why not just skip them and stop trying to handle them in a special way.

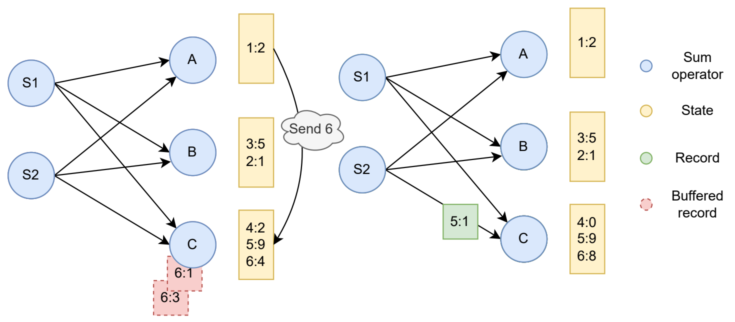
An online state migration paradigm that follows such an efficient "lazy" fetch-on-demand-based approach is presented in Figure 4.1. In it, we can see an example case of state migration taking place between operators, following a reconfiguration command:



(A) Step 1



(B) Step 2



(C) Step 3

FIGURE 4.1: Overview of "Lazy-fetch" state migration process steps.

Step 1 shows normal operation of the system, with data records flowing through and being processed. Sum operators have access to specific key,value pairs while new keyed records that arrive update the existing values using the new ones.

Step 2 follows a reconfiguration command and we can see a record with key 6 now being routed to operator *C*, rather than *A* where it would be routed to before the reconfiguration. As the state is not currently present in operator C, the record is buffered while the fetch command is issued to the operator *A* that holds the associated state. The dataflow is not disturbed and as such new records for the same or other keys can arrive.

Step 3 depicts a number of events taking place simultaneously. First, the record for key 4 that arrived while the fetch operation was ongoing has been processed normally and the corresponding state updated. Second, while the fetch operation was still taking place, an additional record for key 6 arrived on the same operator and was also buffered until the state becomes available. Next to that, operator *A* uninstalled state for key 6 and sent it over to operator *C* who installed it locally. Finally, the necessary state is available and processing of buffered or future records for the same key, can resume.

In terms of the two important aspects mentioned back in Section 2.4.1:

- Route updates: A fetch-on-demand approach can allow processing while the routing updates take place, regardless of whether they have concluded or not. The way the updates take place is of little importance as the proposed solution would not be waiting on it to conclude before continuing with normal operation in any way.
- State transfers: Would only take place on a request basis, as arriving records could ask for non-local state, which would trigger a lightweight migration.

Other than that, costs associated with moving state regarding keys that are not actually being used, would not have to be paid.

Regarding actually "completing" the migration, meaning in fact having the state in the intended worker at some point, this is where the approach differs. State would not be guaranteed to be located in a target worker following a signal for reconfiguration, unless a relevant record requiring that state reaches the target worker. In our view, this is highly beneficial since:

- If a relevant record arrives, then migration costs are paid in a lightweight fashion once and the state subsequently resides with the target operator, requiring no further transfers in the future. Subsequent records for the same key in this operator will utilize the now local state, operating in a steady-state manner.
- If a relevant record does not arrive, then transfer costs will not be paid at all. The state will not be moved from the source operator, until when and if in the future another record arrives requesting it, at which point the migration can in fact take place. This would lead to a more evenly spread-out cost over time, avoiding local spikes following the migration command and in turn only having minor extra latency added at later times.

Consistency guarantees will effectively be borrowed by Meces, as the mechanism is similar and only one operator holds the state, making updates at any given time. The lazy approach can be argued to improve in almost all areas where the previous approaches exhibited shortcomings. It requires no additional steady-state resource usage, no mesh communication networks and removes all costs associated with transferring the states in bulk after *syncing* is completed on all operators and before concluding the migration.

4.2 Fault Tolerance

The intricate relationship between state migration and fault tolerance has already been briefly mentioned back in Section 2, but we do believe that in simple terms a solution about one cannot be viewed without examining the impact it has on the other. As far as our lazy approach is concerned, fault tolerance is in fact improved by dropping the limitation Meces imposed of pausing checkpoints. Now, checkpoints can be triggered at any given point, as long as a special routing log table introduced is also stored (mentioned below in Section 4.3) and any local fetch potentially taking place is awaited. The supporting idea behind this freedom, describes that as long as any operator presented with a record whose state is not local, can find the peer actually holding that state to employ fetch-on-demand, the restore point in relation to the migration process plays no role in consistency. The system can restore from any given checkpoint and continue normal operation.

4.3 Potential Limitations

Naturally, with a radical design choice such as the one where following a state migration command, no migration necessarily takes place, questions for a number of potential limitations arise. These contain use-cases where **potentially** special care or limitations have to be imposed, as with similar systems presented before, to ensure system stability and availability.

The system's design at its core, shifts the cost-associated factor from the number of keys for which state needs to be migrated, to the number of keys that will actually arrive following a migration command. Regardless of whether the system is migrating few or many keys, the latency as a result of migration is now dependent on the records arriving afterwards. A natural first enquiry would be about the case where **all** migrated keys are used right after migration. Even then, we expect the system to exhibit better performance than blocked-based migration because of the order the states become available, as found by Meces and portrayed in Figure 3.6. However, an average case will see the use of only a number of keys, which themselves are presented in varying frequencies (some used more frequently than others) and some not even being used for a while. As a result, gives the lazy approach the best chances in performance since the latency for fetching states for these rare records, would not be detected until a later time or not even at all.

A second potentially limiting use-case, would be scaling-in. Migrating to less workers, contradicts the paradigm of lazy fetch, as there state still resides in source workers until actually needed from target ones. However, since in the case of removing workers from the system these workers will at some point become unavailable, all the states need to be moves even if relevant records do not actually arrive. Penalizing as such an operation might be, there is a silver lining in the fact that scaling-in usually comes as a direct result of reduced workload and less strict latency requirements. Even though higher latency would be observed in this special case, it would be in an otherwise relatively low utilization period. The reason why in contrast, Rhino's corresponding limitation on scale-out scenarios, was viewed as a disadvantage back in Section 3.3.3, is that these typically occur under heavy load. Typically, some form of back-pressure sensing infrastructure will trigger a scale-out reconfiguration to cope with increased loads. Doing so in an inefficient manner means that towering record-processing requests and latency requirements might not be met.

Another point, made when presenting Meces in Section 3.4, is a special case of "ping-pong" that can potentially occur on a fetch-on-demand-based approach. When using distributed routing tables, the intermediate stage immediately following a reconfiguration when they are not all synced, can cause some records to follow the "old" configuration in parts of the dataflow graph and others to follow the "new" one. In the example presented in Figure 4.1, that would mean that after Step 3 has occured, a record with key 6 is still routed to operator A (perhaps because $S1$, unlike $S2$, is not updated), calling for the state to be transferred back to its original owner. This case could potentially incur unnecessary state transfer costs, as future records in this example would potentially dictate the transfer of the same state back to operator C. The possibility of the issue presenting itself, is directly related to the amount of time distributed routing tables require to sync together and is mentioned as potential future work in Section 7. However, we do not consider it as a major limitation as at worst it would introduce a potentially minimal extra latency in affected topologies while at best it could be dealt with by ensuring routing tables are synced before actual states are moved and the reconfiguration in fact takes place.

Finally, an important aspect in making the lazy fetch approach work, is the ability for any target worker to identify the source worker without causing disruptions or exhaustive queries to all peers. An already existing solution is that employed by Meces, in the form of an external KVS, which admittedly requires unnecessarily many messages between operators to complete. Another potential solution, depending on the underlying infrastructure and topology, would be a routing log table, where previous routings of records are logged and can later be used to identify the previous route that a similarly keyed record followed. This way, direct requests can be made between parallel operators to retrieve the state. The state transfer protocol itself can be any type of distributed storage, external KVS or direct TCP connection between workers, again depending on the underlying infrastructure and topology.

4.4 System Implementation

Moving on from a theoretical design approach, to a more practical explanation of the "lazy" approach, we implement the online state migration feature on an existing system. As system topology, capabilities and overall infrastructure highly affect any experimentation necessary, a description of the underlying system follows.

In Figure 4.2, a graphical representation of the architecture of the SPE used can be observed.

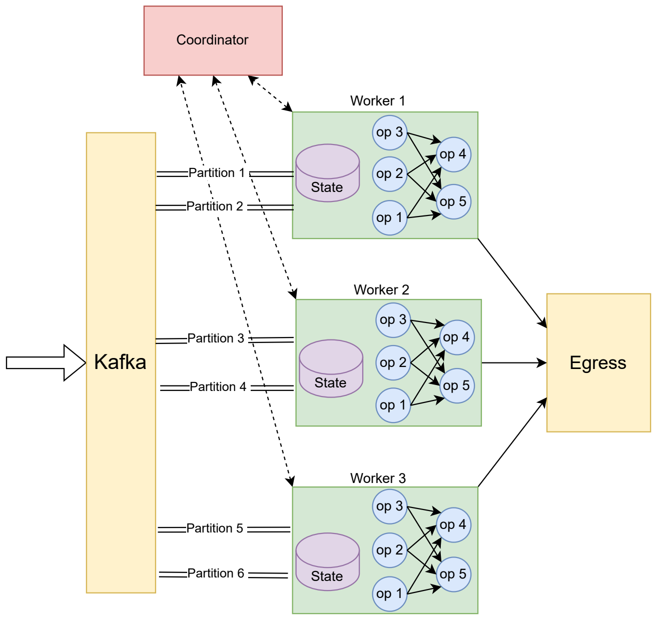


FIGURE 4.2: Experimental SPE system overview.

The most important parts of the system are:

- **Ingress**: Data are introduced to the system through a Kafka installation a client can communicate with, which also connects to the workers available by the system. Kafka partitions are used to increase bandwidth in the ingress-worker communication channel, but also serve parallel instances of the same operators within the same worker and boost performance. Routing takes place at this level, with any reconfiguration commands affecting local routing tables that as a results change how specific keys are routed to partitions and therefore workers and their operators.
- **Workers**: Every worker holds a local replica of the dataflow graph specified by the client. Parallelization of operators either inter-worker or intra-worker is possible, with the latter enjoying shared access to the local state. That means that looking back to Figure 4.2, $op3$ in worker 1 can actually have two copies in the same worker, one reading from partition 1 while the other reads from partition 2, effectively allowing them to work in parallel.
- **Inter-worker communication**: Perhaps most importantly, the system employs worker-to-worker TCP connections for direct command message exchange.

These channels are supported by long-running connections and any message exchange subsequently goes through a serialization/de-serialization process.

- **Coordinator**: A central service is used to coordinate system-wide actions such as submitting the client's dataflow graph, commencing fault-tolerance processes in the face of failures and identifying system state at any given time.
- **Egress**: Processing results are written back to Kafka, using dedicated partitions, not affecting the data processing latency of the system.

For our "lazy" approach implementation, we retain a routing log table before assigning Kafka partitions to records, so as to detect at any given point where the key was last routed to-and subsequently who holds the state for that key-. As a result, an operator trying to process a record whose key is not local to the worker, can identify the source worker and begin a state migration process as described in Figure 4.1. For asking and delivering the state, the inter-worker connection is used, so as to query the source worker directly and not disturb the dataflow by injecting special markers. State migrations in the system occur in a key-level granularity, so as to reduce the volume being transferred through the inter-worker channels and limit the "ping-pong" event, as described back in Section 4.3. For buffered in-flight records during the fetch operation, Python's asyncio events are used to both buffer and then signal normal processing, once the state is available.

5 Evaluation

Next, we perform an evaluation on our implementation of the "lazy" approach, through a series of experiments looking into different important aspects that can affect performance. We try to quantify the solution's efficiency when dealing with various demanding scenarios, in order to pinpoint its' strong points or potential drawbacks in practice.

In these scenarios, we start with normal processing under varying conditions and after 3 seconds of operation we manually trigger a re-shuffling of a specified amount of keys, among the existing workers. In practice, we re-assign a number of keys from a source worker to any of the others, now called target workers. This testing process reflects a potential load-balancing or code update use-case in a relevant system. There is no reason to believe that a scale-out test would behave differently in any way, since the same logic would hold, probably with even better post-migration performance since new resources would be introduced to the system. For scaling in, special care would be needed to deal with the fact that state cannot be left behind in a worker going offline, as noted already in Section 4.3.

For measurements, we look at end-to-end latency measured with a timestamp first set when the record sent by the client first enters Kafka (ingress) and compare that with a second timestamp set when the output of the same record is written to Kafka as output (egress). The difference between the two, provides a per-record latency result, which we group under time of execution to provide an overview of the system's performance over time. To that end, every scenario runs for 10 seconds and the results are grouped under the second of execution at which the record was generated. To report the result of the group, we use the 50^{th} and 99^{th} percentiles of the values recorded to represent average and worst-case performance of the solution at test. Since when migrating for instance, the records of keys being migrated might exhibit worse performance that those not taking place in the migration, looking into the average performance is not enough and the worst-performing part of the system needs to be highlighted.

5.1 Baseline

To provide us with a point of reference, since the underlying system's performance cannot be directly compared to Flink for example, we designed a baseline online state migration solution. The baseline solution in essence follows the partial-pause-and-resume design paradigm and following the reconfiguration command, performs the following steps:

1. The coordinator (seen in Figure 4.2), notifies all target workers of incoming states.

2. All workers notified of receiving state from peers, start buffering arriving records, until all source workers have successfully sent the corresponding states.

3. After a worker has received all states that it was notified to expect, it can stop buffering and start processing buffered or any new arriving records.

Since the first step in this process can take some time and relevant records for keys being migrated can arrive at that point, we give the baseline solution an advantage by "blocking" ingress until the coordinator has notified all workers. This is done to ensure correct processing and essentially gives the baseline an edge since in the meantime no "start" timestamps are logged for the records and no latency is recorded. After the notifications are sent, the workers can start buffering if necessary, transferring states between them and ingress continues normal operation.

For transferring the states between workers, the direct communication channels used by the system are employed, whereas buffering again is handled through asyncio events. Whereas in the case of the lazy-fetch approach, no additional steps need to be taken other than updating the routing tables and passing through the routing log information, in the baseline case, we make an extra call on the client side to signal the coordinator, which commences the migration process.

5.2 Experimental Setup

To actually execute the experiments that follow, the machine used consists of an AMD Ryzen 37000x CPU with 8 cores/16 threads and 32GB of DDR4 RAM @ 3600MHz. The environment used is a Linux Ubuntu 22.04 distro, running Python 3.10. The deployment of the Kafka, coordinator and 4x worker services were achieved through docker containers using 1 CPU core and 1GB of memory each.

For the client side, we run a script on a dedicated thread with an input rate of **4,000 records/sec** for **10 seconds** total. The operator at hand is a cut-down version of YCSB by Barata, Bernardino, and Furtado, 2014. In it, we first create entries for **1,000,000** keys by setting a starting value and then we use the client to generate an appropriate number of random keys every second, upon which we call simple update functions that just read the corresponding value and increase it by 1. This way, we can keep track of the keys generated and check the output for consistency and completion.

For the operator used, we selected **8 partitions**, which are automatically assigned by the system in a round-robin fashion. That means that each worker ends up with 2 partitions of the same operator, running in parallel for different keys. The selection of operator partitions and workers' number was such as to be bearable by the test machine and not present external delays due to insufficient resources.

As mentioned already, for measuring the end-to-end latency, we create unique identifiers and assign records a timestamp once they are submitted to ingress and again in egress. Since the same host machine runs for all workers, the timestamps can be considered consistent and latency measured as the difference between the two. The measurements are presented as per second of execution, referring to the second of execution at which they were first presented to ingress, rather than the one they were produced to egress. That means, that a record generated at the 3^{rd} second, but due to increased latency took 2 seconds to be produced to egress, will still be presented at the 3^{rd} second, albeit with the observed latency of 2 seconds.

5.3 Number of keys experiment

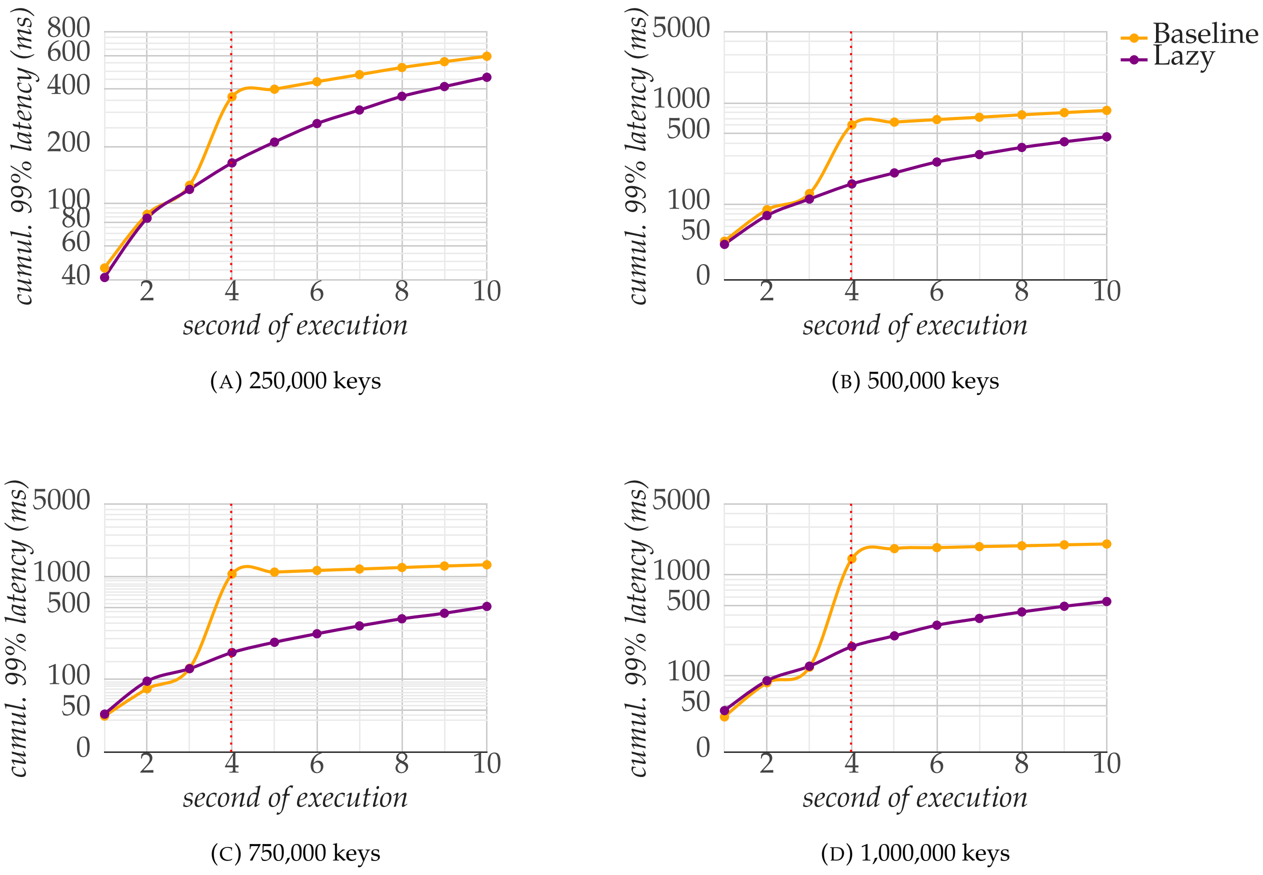
For the first experiment, we look into the effect the amount of state, based on number of keys, has on end-to-end latency when migrating. To achieve this, we select a specified amount of random keys to be migrated, for which we alter the worker assignment. The steps of the test are as follows:

1. Assign 1,000,000 keys to workers evenly and initialize their values.
2. For seconds 1 to 3, generate random keys and request update functions on corresponding workers.
3. At the 4^{th} second, for the pre-selected number of random keys, change worker assignment to any other. The moment is marked in the figures as a vertical dotted red line.
4. Continue generating keys and making requests to the system.
5. Measure and report performance.

For the amount of keys tested, given the fact that our system was initialized with 1 million keys, we look into migrating 25%, 50%, 75% and 100% of the keys. To provide a direct overview of observed performance, we present a cumulative 99^{th} percentile graph of the end-to-end latency per second of execution in Figure 5.1. The 99^{th} percentile provides a closer look into the peak values, bound to be associated with keys that are being migrated, or buffered and suffer increased latency as a result. Being a cumulative graph, an ideal solution should exhibit not distinguishable "jumps" in latency and an overall linearly increasing line, as end-to-end latency remains constant.

We can clearly see how the baseline solution fails to do so, suffering a clear increase in latency at the critical point of the reconfiguration and as a result finishing off with poor performance overall. Moreover, the latency increase seems to worsen as the number of keys being migrated itself increases. On the other hand, the lazy approach seems to be relatively unphased by the reconfiguration call, whereas the performance gap with the baseline solution increases, as more keys are being migrated. The accumulated value at the end (10^{th} second of execution) tells the story of a big performance improvement at system-level, when using the lazy approach over the baseline.

To dive deeper into the performance numbers, we also look into the actual 99^{th} percentile latency in Figure 5.2. The results show how bad the worst performing keys really did, telling a similar story to before. The baseline solution exhibits big increases in latency following the reconfiguration, further worsening as the number of keys -and therefore the amount of state- marked for migration increases. Normal (or ready-state) system performance seems to hover around the 20ms point, while the "jump" ranges from 50ms all the way up to 1,200ms, an increase of 2 order of magnitude compared to ready-state performance. Interestingly enough, the latency increase in cases (C) and (D), effectively "spills" into the next second of execution, causing increases there as well. This phenomenon can be easily explained as the lengthy buffering in this extreme case, apart from the halt in system execution, resembles a massive increase in input rate as buffered records keep accumulating, leaving the system to cope with processing all these records at once, following the transfer. Normally, our input rate of 4,000 records/second is much below the system's throughput threshold, but the buffering necessary starts stressing the system limits. Adversely, the lazy approach is confirmed to not exhibit any negative changes in performance and in fact retains a much stable performance throughout. In some cases we can see the baseline performing slightly better post-migration, explained by the fact that the lazy approach is in fact performing minor migrations at that



(A) 250,000 keys

(B) 500,000 keys

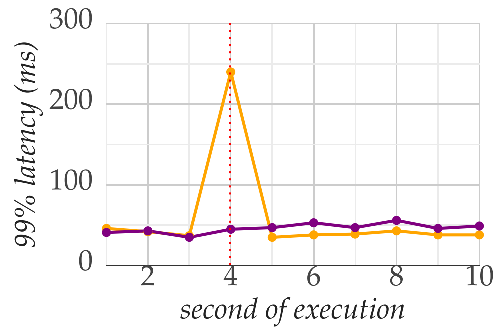
(C) 750,000 keys

(D) 1,000,000 keys

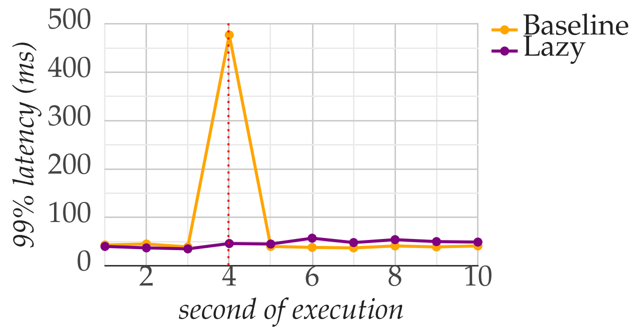
FIGURE 5.1: Cumulative 99th percentile of end-to-end latency, by second of execution. Reconfiguration triggered at the 4th second.

point, however it still succeeds in staying within a very acceptable 10ms difference to ready-state performance.

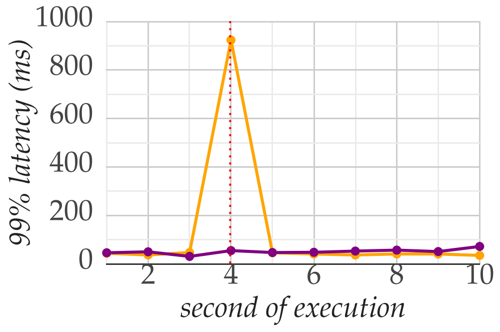
Looking into the 50th percentile numbers in Figure 5.3, tells a similar story to before. This time we are presented with the average performance of the system, but the overall image remains the same. The baseline solution performs much worse following the reconfiguration, exhibiting a performance drop of over 2 orders of magnitude, while the lazy remains very stable. We can see the average performance drop of the system also transcending the second of execution and "spilling" into the next ones on the last case, indicating that this is a system-wide phenomenon, noticeable in its overall performance, rather than just contained to a number of keys performing worse that others.



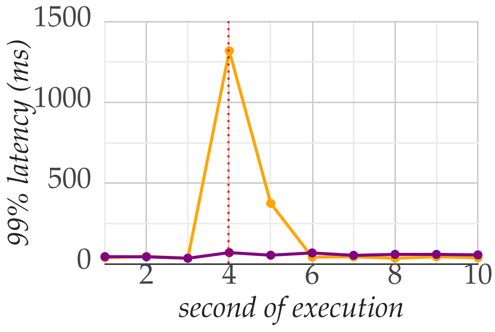
(A) 250,000 keys



(B) 500,000 keys

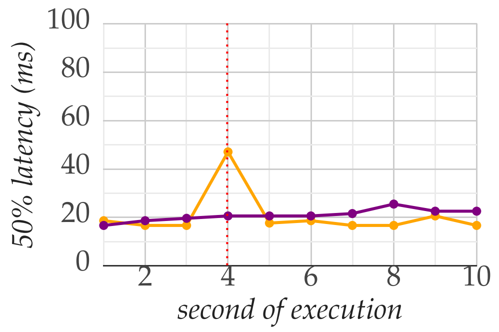


(C) 750,000 keys

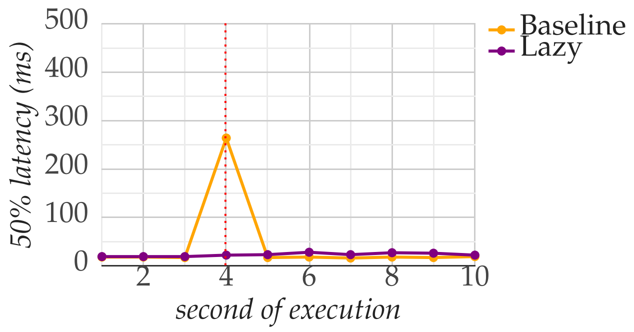


(D) 1,000,000 keys

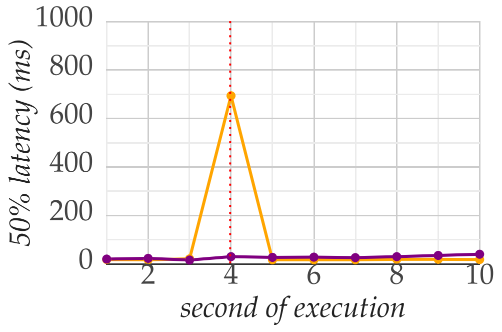
FIGURE 5.2: 99th percentile of end-to-end latency, by second of execution. Reconfiguration triggered at the 4th second.



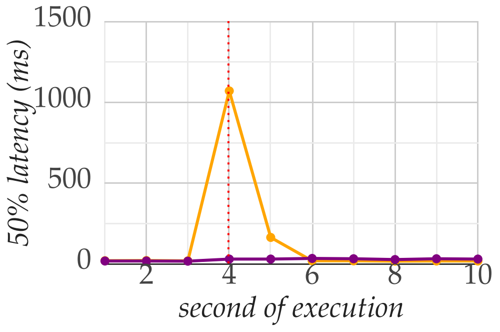
(A) 250,000 keys



(B) 500,000 keys



(C) 750,000 keys



(D) 1,000,000 keys

FIGURE 5.3: 50th percentile of end-to-end latency, by second of execution. Reconfiguration triggered at the 4th second.

5.4 State size experiment

The second experiment expands upon the first one, by also examining the effects of state size in memory occupied, rather than just the number of keys being migrated. Since the main source of delays in state migration is network transfers that require serialization/de-serialization, synchronization and transfers themselves, looking into cases where these partial processes are further hindered, can provide invaluable insights into the two implementations' performances.

The testing process is similar to before, although this time we gradually increase the "packet size" associated with each key, effectively increasing the state size without having to actually store more data. In essence, we are further penalizing the state transfer operations, potentially favouring more efficient decision to migrate.

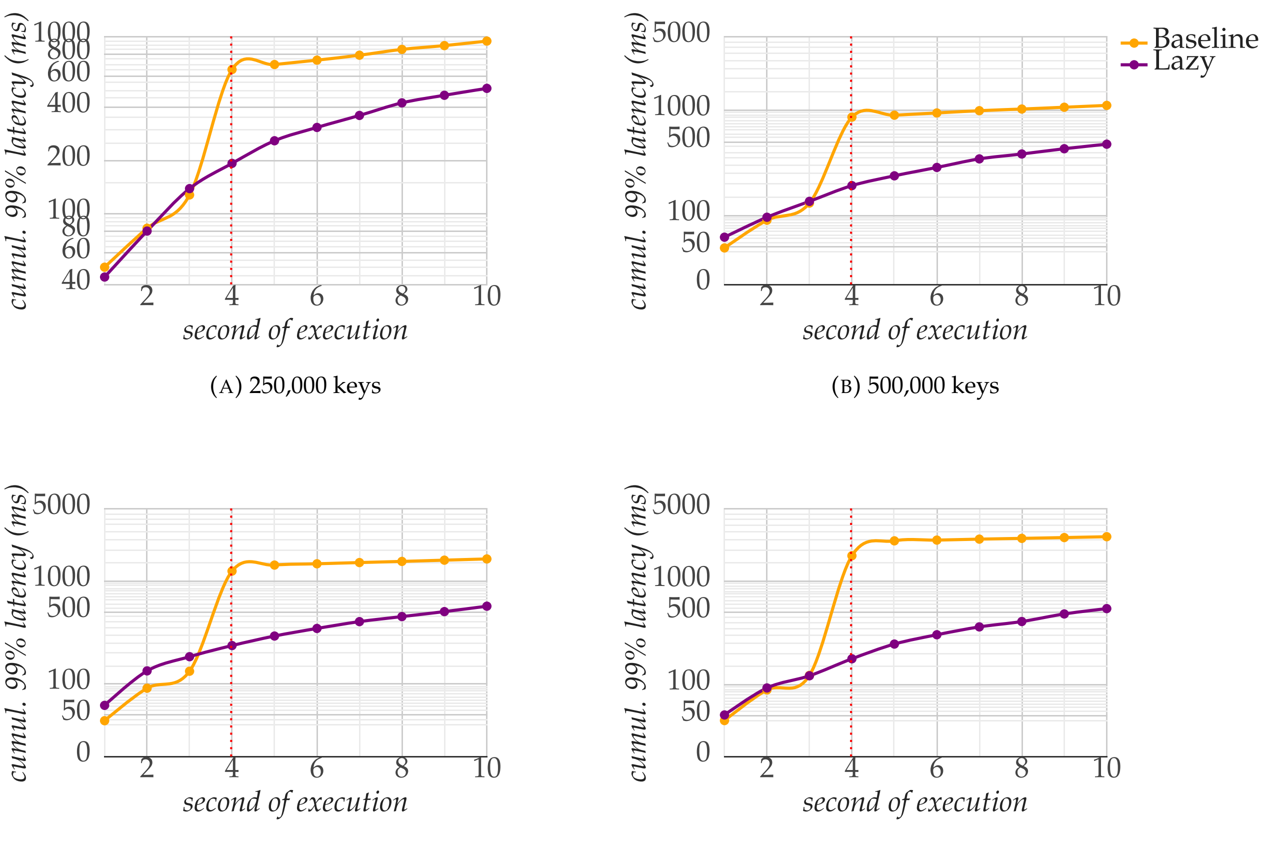
An overview of the measurement results, showed a similar behaviour for both systems in 50^{th} and 99^{th} percentile numbers and in order to avoid redundancy, we decided to focus on reporting and explaining the latter.

First, we increase the state size by 10 times from 50MB up to 500MB, looking into the same amount of keys being transferred as before. The performance overview is presented in Figure 5.4. The behaviour remains the same, with the baseline dealing with the migration in a less efficient manner than the lazy approach and dealing with scaling workloads even worse. Again, the lazy approach presents very stable performance, a much sought-after characteristic of relevant systems in corresponding operations. Overall, the accumulated latency at the end, is throughout an order of magnitude better in favour of the lazy approach.

Looking into the system's performance for each second of execution in Figure 5.5, we can clearly spot the same "jump" in end-to-end latency at the moment when the migration is triggered and once more at case (D), the next second of execution is affected. Not only is the latency at the 5^{th} second of execution increased, but it almost reaches the same levels as the moment of migration itself, hinting at the fact that were this situation to get any worse, another second would also be affected. As expected, the two systems behave similarly before the migration is triggered, while the lazy approach only shows very minor overhead at later times when further migrations are taking place.

Next, we increase the state size even more, this time by 100 times from 50MB to 5GB, in an effort to further expose the weak or strong points of either systems. The cumulative results in Figure 5.6 succeed in showing just that. This time, the state size is simply too big for the baseline solution to efficiently move while dealing with incoming records, leading to a substantial performance drop. The gap in performance between the two solutions is further increased as the number of keys itself increases, peaking at a staggering 3 orders of magnitude difference in accumulated latency! At this point, the performance demonstrated by the baseline solution can be characterized as problematic for the underlying system, as several requirements of a modern SPE are not satisfied. In contrast, the lazy solution deals with the increased workload with minimal, if any, overhead introduced to the system.

Focusing once more on the performance per second of the system, Figure 5.7 finds the latency "jumps" are even more exaggerated this time, showing delays in the processing time of almost a minute long! In contrast, the lazy approach stays in the tenths to low hundreds of milliseconds of latency in processing time, while the migration trigger itself at the 4^{th} second, does not cause any major disruption in the performance observed. As predicted, the performance degradation in the case of the baseline solution is such that it takes the system an increasing amount of time before performance returns to pre-migration levels. In fact, it completely fails to do so in



(A) 250,000 keys

(B) 500,000 keys

(C) 750,000 keys

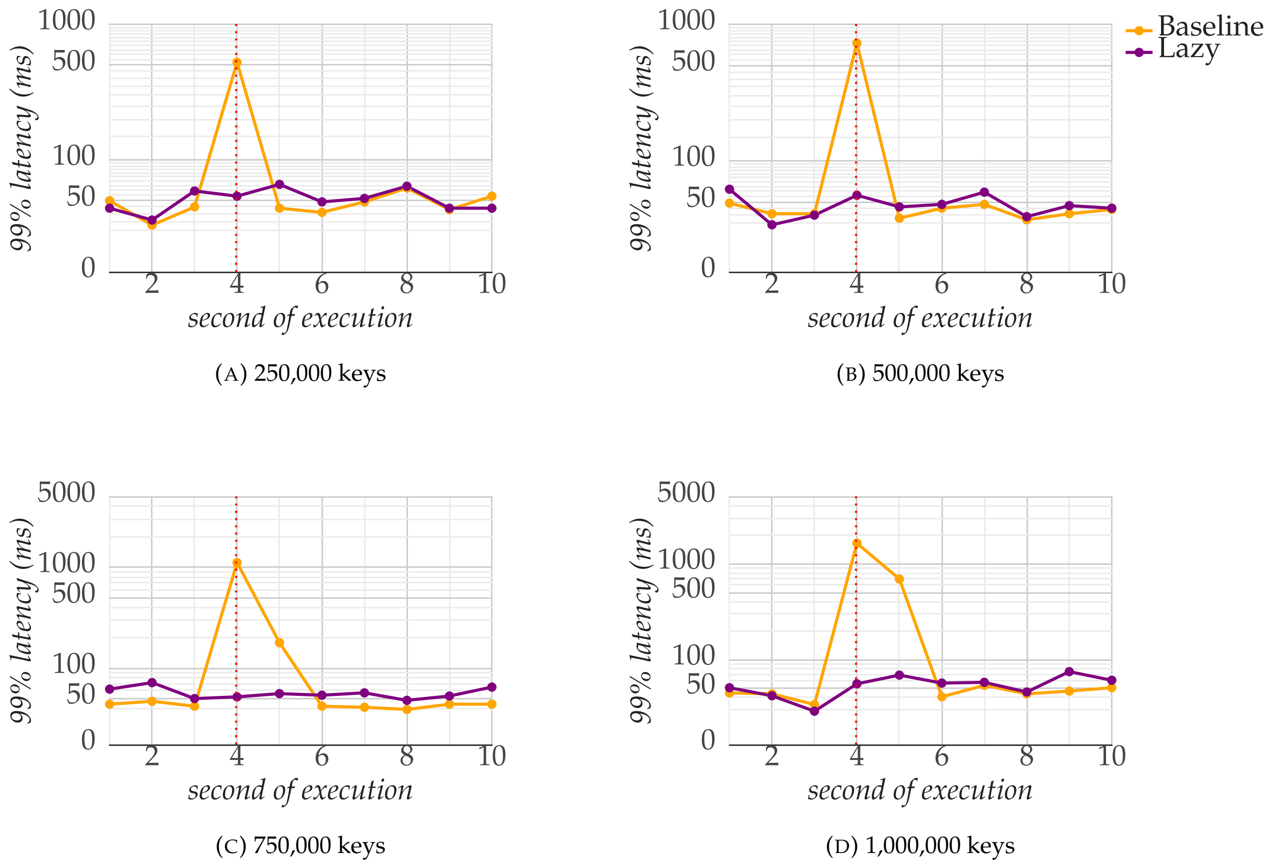
(D) 1,000,000 keys

FIGURE 5.4: Cumulative 99th percentile of end-to-end latency, by second of execution. State size increased 500MB. Reconfiguration triggered at the 4th second.

the more extreme cases in graphs (C) and (D), in the provided 10 seconds of total execution time.

The severe performance degradation exhibited by the baseline solution can be explained by the fact that since all the specified keys and assorted values are transferred at once, the increase in size causes an already big object needed to be moved, to become even bigger. This compound increase directly affects serialization and deserialization overhead in workers and network delays when transferring. This situation keeps getting worse, as the number of keys selected for migration increases, until the object size is simply too big to perform the task at hand.On the other hand, the lazy approach only requires state transfers on keys for records arriving on corresponding workers, while the transfers themselves happen on a key-level granularity, leading to much less transfer overhead.

Surprisingly enough, the baseline solution never succeeds dealing with state sizes past the 5GB mark, triggering the underlying system's fault tolerance mechanisms, as workers work too hard and for too long trying to cope with the difficult task at hand. On the other hand, the stable performance profile displayed by the lazy approach persists. In fact, by keeping the selection of keys to be migrated to 250,000, it's not before we increase the state size all the way up to 5TB, before we see a noticeable performance drop.



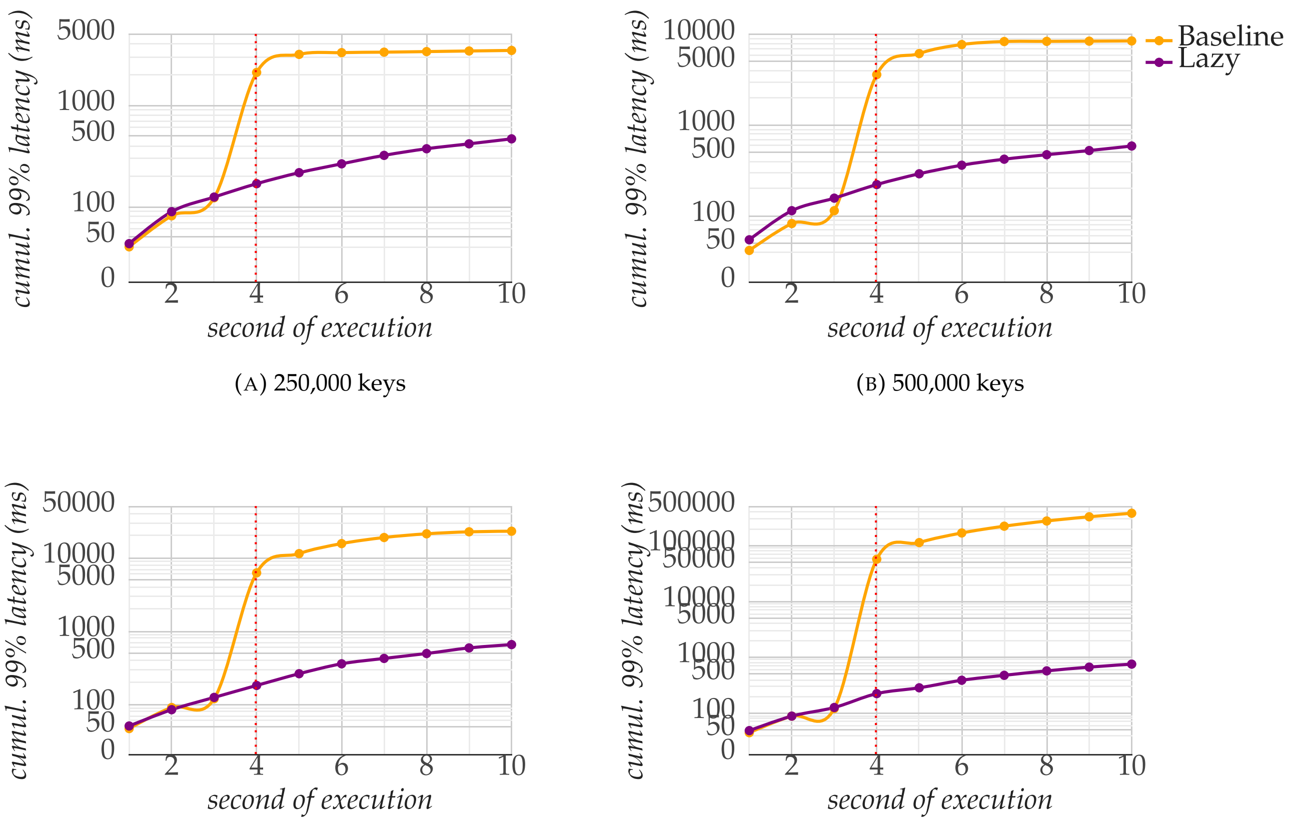
(A) 250,000 keys

(B) 500,000 keys

(C) 750,000 keys

(D) 1,000,000 keys

FIGURE 5.5: 99th percentile of end-to-end latecy, by second of execution. State size increased to 500MB. Reconfiguration triggered at the 4th second.



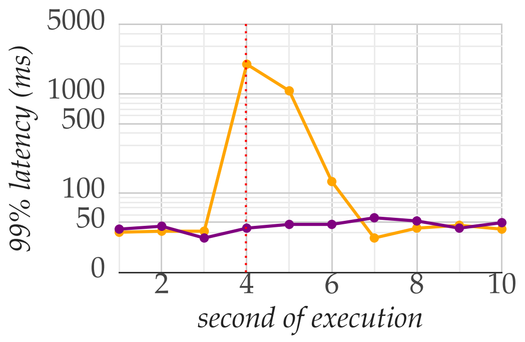
(A) 250,000 keys

(B) 500,000 keys

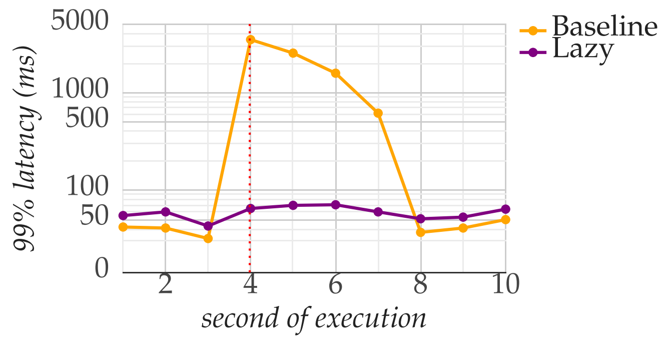
(C) 750,000 keys

(D) 1,000,000 keys

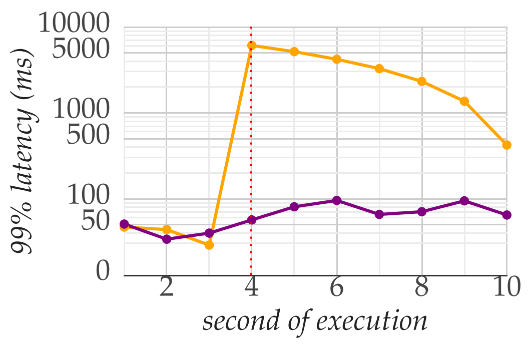
FIGURE 5.6: Cumulative 99th percentile of end-to-end latecy, by second of execution. State size increased to 5GB. Reconfiguration triggered at the 4^{th} second.



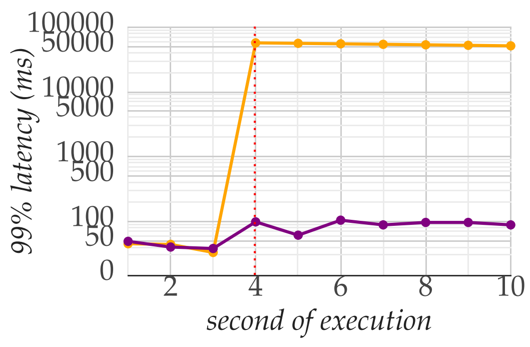
(A) 250,000 keys



(B) 500,000 keys



(C) 750,000 keys



(D) 1,000,000 keys

FIGURE 5.7: 99th percentile of end-to-end latecy, by second of execution. State size increased to 5GB.Reconfiguration triggered at the 4th second.

5.5 Record distribution experiment

The main novelty that the lazy solution brings to the table, revolves around the causality effect between state migrations and the records flowing through the system, following a reconfiguration command. As such, any benefits displayed by the system employing the lazy solution, stem from the fact that it does not need to "pay" migration costs unless necessary. In fact, taking a closer look into the results available from the previous 2 experiments, reveals that the lazy approach only ever migrates $2,75\%$ to $2,8\%$ of the keys selected for migration, at any given case. The performance difference observed so far then, mainly comes down to the fact that the lazy approach migrates over 30 times less state compared to baseline! Not only that, but the transfers themselves are not performed at the exact moment of the migration, but can be triggered at any point when the corresponding record first arrives.

To that end, we devise an appropriate experiment, examining this causality effect closer. So far, we generated keys for the records to be processed by the system, using a uniform distribution. In order to test the causality effect, we employ a zipfian distribution, gradually increasing the factor and measuring the effect it has on performance. Simply put, every successive increase on the zipfian factor, increases the chances of generation for specific keys, meaning that at the end only a handful of keys are generated and flow through the system, as opposed to all of them having an equal chance.

We expect that such skewed workloads will further benefit the fetch mechanism behind the lazy approach. Thus, we select the extreme case mentioned at Section 5.4 and operate with a state size of 5TB, while moving 250,000 keys in the reconfiguration command. This way, we have a badly performing workload in our hands, where the effects of the zipfian distribution can be more easily observed and possibly even beneficial.

In Figure 5.8 we observe how the migrations following the command at the 4^{th} second of execution, have now a more profound effect and in case (A) where we still have basically a uniform distribution, take the end-to-end latency of the system from 20ms to 300ms. Since now these transfers require a significant amount of processing themselves, they are not as lightweight as before. However, increasing the factor seems to have a beneficial effect on the system's performance. Going from a uniform distribution, to a slightly skewed one in case (B) drops the 50^{th} percentile of end-to-end latency at around 200ms, while further increasing the factor at (C) drops it to an even better 100ms. The extreme case at (D), offers some interesting insight as the end-to-end latency at the moment of the reconfiguration command is higher than before, but drops even lower than the 100ms mark afterwards. In order to better understand why this happens, we first need to also look into the 99^{th} percentile and analyze the latency on worst-performing keys, as now specific keys can be more "important" than others.

Figure 5.9 offers a detailed view on this matter. The graphs show a progressive improvement, with the factor selection at 0.6 exhibiting the best performance overall. The reason behind the improvement can be easily explained when looking into the logs. Using a zipfian distribution with a factor of 0.3 results in only 2% of the keys being transferred, while increasing the factor to 0.6 brings the same number down to just 1.5%. Less transfers mean less buffering and transfer costs associated, which in turn translates to less latency overhead introduced to the system, when keeping the amount of keys transferred stable.

However, the same extreme case at (D) paints a much different picture. The latency "jump" at the moment of the migration is very high, at 1500ms and gets even

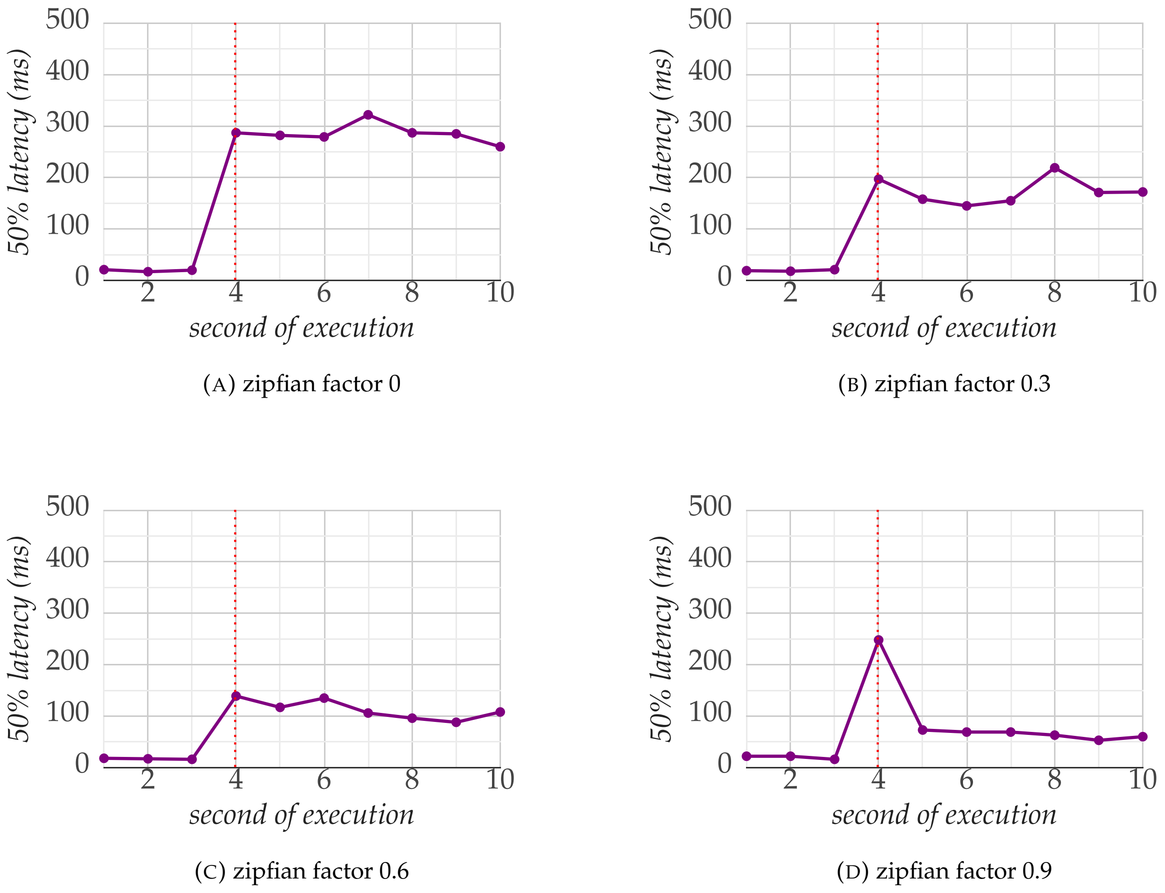
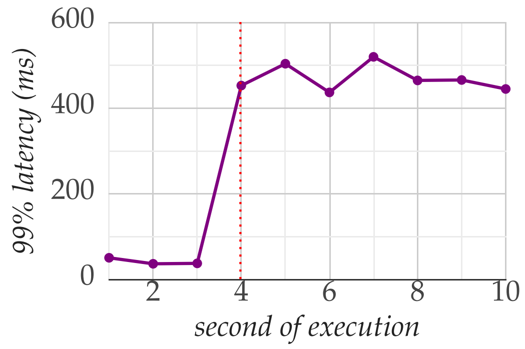


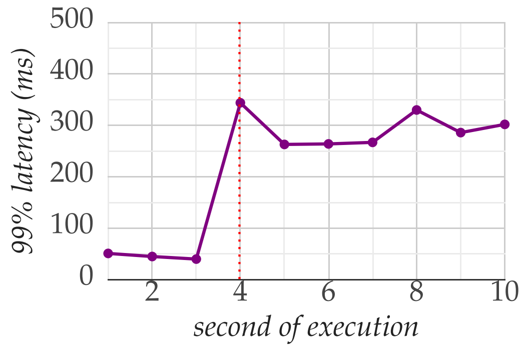
FIGURE 5.8: 50th percentile of end-to-end latecy using different record distributions, by second of execution. State size is increased to 5TB. Reconfiguration triggered at the 4th second.

worse at time progresses. The fact that the same behaviour is not directly observable at Figure 5.8, indicates that the issue is located at these high-frequency keys that are now mainly flowing through the system. However, it still fails to explain the previously observed "jump" in end-to-end latency.

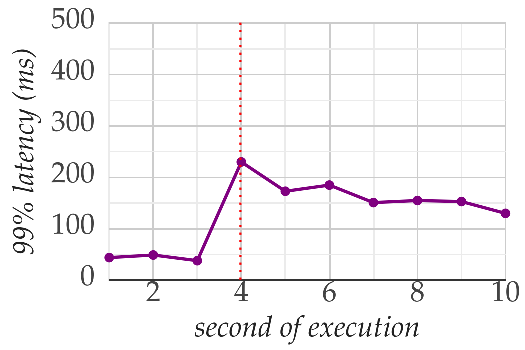
To shed more light into the issue, we ran the same experiment on the initial state size, to factor out the big state size. Figure 5.10 shows the system coping with the migration without noticeable spikes, in all zipfian factor selections. However, Figure 5.11 fills the missing piece of the puzzle, showing that even though the first 3 cases are also stable, case (D) with the highest zipfian factor creates problems in the system. In fact, the latency spikes occur even before any migration is triggered and even get better following it. This behaviour supports the hypothesis that the underlying system struggles under extremely skewed workloads. Even though the same skeweness favours the causal migration decisions our lazy approach takes, it directly affects effective sharding in the system, creating heavily imbalanced loads that on their own create performance issues, unrelated to the state migration process we are studying. As such, the negative impact highly skewed workloads have on the system at test, depends more on the underlying system and less on the state migration solution at test.



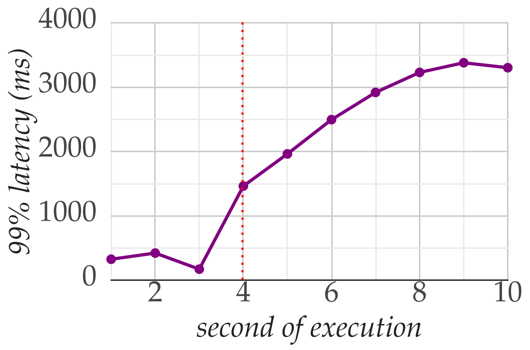
(A) zipfian factor 0



(B) zipfian factor 0.3

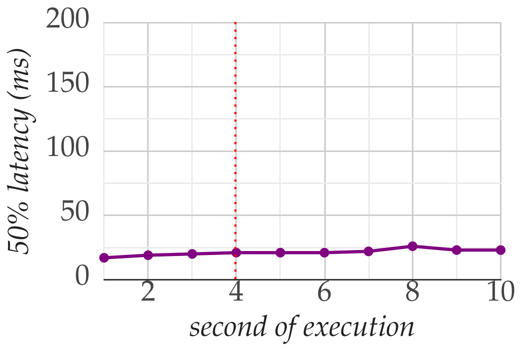


(C) zipfian factor 0.6

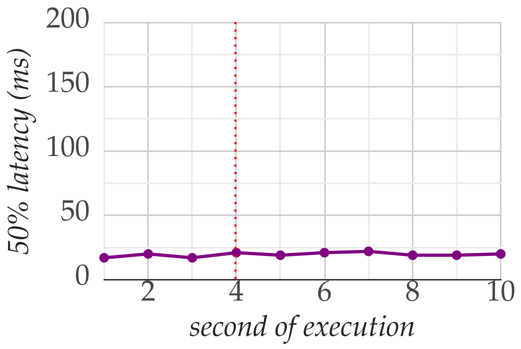


(D) zipfian factor 0.9

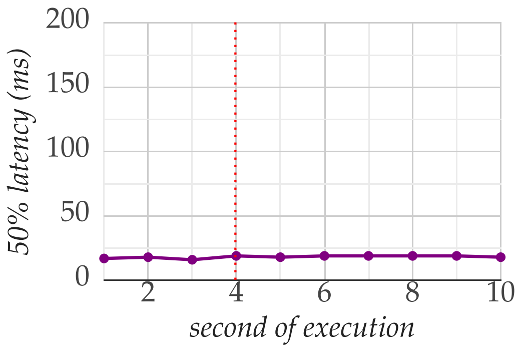
FIGURE 5.9: 99th percentile of end-to-end latecy using different record distributions, by second of execution. State size is increased to 5TB. Reconfiguration triggered at the 4th second.



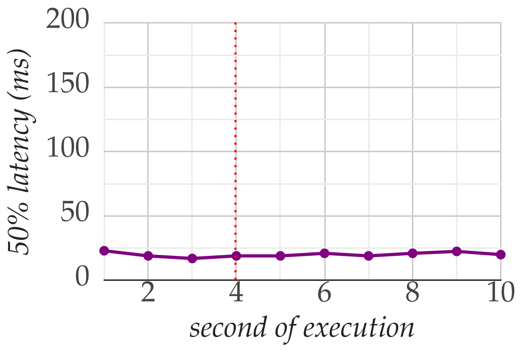
(A) zipfian factor 0



(B) zipfian factor 0.3

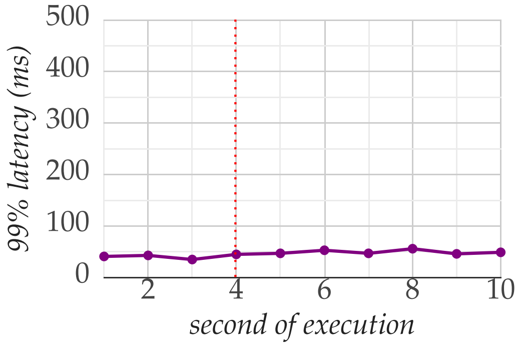


(C) zipfian factor 0.6

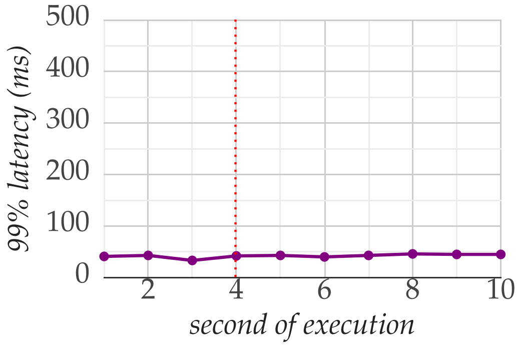


(D) zipfian factor 0.9

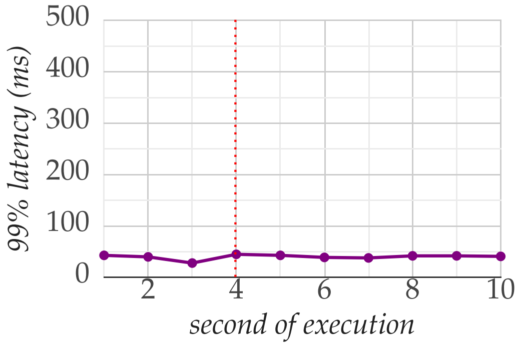
FIGURE 5.10: 99th percentile of end-to-end latecy using different record distributions, by second of execution. Original state size of 50MB. Reconfiguration triggered at the 4th second.



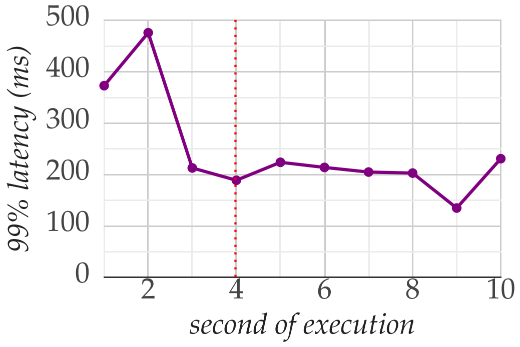
(A) zipfian factor 0



(B) zipfian factor 0.3



(C) zipfian factor 0.6



(D) zipfian factor 0.9

FIGURE 5.11: 99th percentile percentile of end-to-end latecy using different record distributions, by second of execution. Original state size of 50MB. Reconfiguration triggered at the 4th second.

5.6 Discussion

The experiments presented in this section, focus on the state migration part of an example workload, while examining the impact of the most important factors. Most importantly, we try to assess the impact increasingly harder to process workloads have on the performance of the two systems at test, while also looking into how well their performance scales in relation to the problem's complexity.

We find that the baseline solution, based on the existing SPE, performs poorly under increasing workloads. In times, it introduces several orders of magnitude higher latency to the system, much outside the acceptable range for such systems, while others it even fails to complete executions altogether.

On the other hand, our proposed lazy-based implementation, fairs much better across the board. It succeeds in remaining relatively unphased in the face of the same workloads, while it displays stable performance when the critical reconfiguration commands are issued. Most importantly, results across all experiments find the lazy approach in the millisecond level for end-to-end latency, very close to ready-state performance.

6 Conclusion

In this paper, we visited the problem of online state migration is modern SPE architectures and tried to:

1. Pinpoint the most important steps or aspects included in the process itself, in order to assess solutions around the same basis.
2. Formally state the challenges, requirements and limitations of the problem.
3. Design, implement and evaluate an appropriate solution.

By looking into earlier attempts, we were able to detect design choices that lead to common limitations in existing system, whereas focusing on novelties and important ideas we succeeded in designing and implementing a completely new solution. Focus on the high cost associated with state transfers, while stream processing happens uninterruptedly, lead as far away from trying to move all of the needed state at the moments where reconfigurations are performed. Moreover, earlier studies on the advantages of finer granularity on state transfers, lead us to conclusively use key-level transfers. Finally, the benefits presented with record-first fetch strategies, completed the picture of our novel design and resulted in a feasible solution.

Our solution, outperforms the baseline in all tests performed, while importantly being based on the exact same SPE. The migration mechanism introduces no latency penalties when used, while heavier workloads are dealt with effectively. The solution improves upon the causality effect between records and state transfers and offers more manageable and evenly spread-out costs. The system is therefore able to respond to potential scale-out scenarios much more effectively, offering increased availability at critical times.

Finally, our solution achieves greatly reduced migration costs, without affecting ready-state behaviour. It introduces no extra overhead and no increased resource usage. Fault tolerance is in theory also not affected, as the system makes no special adjustments to existing processes, other than the need to store any routing log tables available.

7 Future Work

As the lazy fetch approach brings a completely new design on state migration processes, a number of associated aspects remain unvisited by our work. Next, we present such ideas or areas of interest.

First, the important part of fault tolerance and the effect a lazy fetch-based solution has on it must be practically explored. Although our work covers fault tolerance on a theoretical level, actual experimentation is necessary to conclusively prove the relation. Inversely, an effectively state migration mechanism can prove invaluable when designing fault-tolerance mechanisms. If state can be moved fast and efficiently between nodes, then failing nodes can use the same mechanism to move necessary information to healthy ones or fetch it back from them during recovery. While consistency guarantees would need to be closely examined, the intricate relationship between the two could lead to the improvement of one resulting to an improvement of the other.

As presented and mentioned in the work by Hoffmann et al., 2019, the granularity at which state transfers happen, can provide a useful trade-off between processing latency and throughput. While in our evaluation's system topology state transfers did not affect the performance gravely, other topologies suffer from this issue much more. In those cases, key groups or gradual migrations can provide a middle ground between heavy migration costs and their effect on our proposed solution must be further explored.

Back in Section 4.3, we mentioned the special case of scaling-in and how a whole-state migration process would need to be employed, as the lazy fetch solution is not an option. Expanding upon the idea, the lazy fetch solution can be coupled together with secondary strategies, to benefit processing down the line. For instance, our solution still leaves un-migrated state back in source nodes, even long after the initial reconfiguration command, in case the associated records never passed through the system. As this could potentially prove problematic when further reconfigurations happen in the future, the migration of this "left-behind" state can be scheduled at times of reduced workloads in the system by an appropriate mechanism. In essence, the lazy fetch mechanism would be used to allows for fast scaling and little extra costs in times of increased workloads and the subsequent scheduling would ensure no left-over tasks affect performance in the future.

Bibliography

Barata, Melyssa, Jorge Bernardino, and Pedro Furtado (2014). “Ycsb and tpc-h: Big data and decision support benchmarks”. In: *2014 IEEE International Congress on Big Data*. IEEE, pp. 800–801.

Castro Fernandez, Raul et al. (2013). “Integrating scale out and fault tolerance in stream processing using operator state management”. In: *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pp. 725–736.

De Matteis, Tiziano and Gabriele Mencagli (2017). “Parallel patterns for window-based stateful operators on data streams: an algorithmic skeleton approach”. In: *International Journal of Parallel Programming* 45.2, pp. 382–401.

Del Monte, Bonaventura et al. (2020). “Rhino: Efficient management of very large distributed state for stream processing engines”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pp. 2471–2486.

Gu, Rong et al. (2022). “Meces: Latency-efficient Rescaling via Prioritized State Migration for Stateful Distributed Stream Processing Systems”. In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 539–556.

Gulisano, Vincenzo et al. (2012). “Streamcloud: An elastic and scalable data streaming system”. In: *IEEE Transactions on Parallel and Distributed Systems* 23.12, pp. 2351–2365.

Hoffmann, Moritz et al. (2019). “Megaphone: Latency-conscious state migration for distributed streaming dataflows”. In: *Proceedings of the VLDB Endowment* 12.9, pp. 1002–1015.

Katsifodimos, Asterios and Sebastian Schelter (2016). “Apache flink: Stream analytics at scale”. In: *2016 IEEE international conference on cloud engineering workshop (IC2EW)*. IEEE, pp. 193–193.

Kroß, Johannes and Helmut Krcmar (2016). “Modeling and simulating Apache Spark streaming applications”. In: *Softwaretechnik-Trends Band 36, Heft 4*.

Mai, Luo et al. (2018). “Chi: A scalable and programmable control plane for distributed stream processing systems”. In: *Proceedings of the VLDB Endowment* 11.10, pp. 1303–1316.

Rajadurai, Sumanaruban et al. (2018). “Gloss: Seamless live reconfiguration and reoptimization of stream programs”. In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 98–112.

Sakr, Sherif, Anna Liu, and Ayman G Fayoumi (2013). “The family of mapreduce and large-scale data processing systems”. In: *ACM Computing Surveys (CSUR)* 46.1, pp. 1–44.

Silvestre, Pedro F et al. (2021). “Clonos: Consistent causal recovery for highly-available streaming dataflows”. In: *Proceedings of the 2021 International Conference on Management of Data*, pp. 1637–1650.

Tucker, Pete et al. (2008). “Nexmark–a benchmark for queries over data streams (draft)”. In: *Technical report*.

Volnes, Espen, Thomas Plagemann, and Vera Goebel (2022). “To Migrate or not to Migrate: An Analysis of Operator Migration in Distributed Stream Processing”. In: *arXiv preprint arXiv:2203.03501*.

Wu, Yingjun and Kian-Lee Tan (2015). “ChronoStream: Elastic stateful stream computation in the cloud”. In: *2015 IEEE 31st International Conference on Data Engineering*. IEEE, pp. 723–734.

Zhang, Shuhao, Juan Soto, and Volker Markl (2022). “A Survey on Transactional Stream Processing”. In: *arXiv preprint arXiv:2208.09827*.